Xavier Leroy

Pierre Weis

MANUEL DE RÉFÉRENCE DU LANGAGE CAML

Table des matières

Introduction

Cet ouvrage contient le manuel de référence du langage Caml et la documentation complète du système Caml Light, un environnement de programmation en Caml distribuée gratuitement. Il s'adresse à des programmeurs Caml expérimentés, et non pas aux débutants. Il vient en complément du livre *Le langage Caml*, des mêmes auteurs chez le même éditeur, qui fournit une introduction progressive au langage Caml et à l'écriture de programmes dans ce langage. Le présent ouvrage s'organise comme suit :

- La partie I, « Manuel de référence du langage Caml », documente précisément la syntaxe et la sémantique du langage Caml.
- La partie II, « Manuel d'utilisation du système Caml Light », explique comment utiliser les outils fournis par le système Caml Light, un environnement de programmation en Caml.
- La partie III, « La bibliothèque standard du système Caml Light », décrit les fonctions offertes par cette bibliothèque.
- La partie IV, « Annexes », explique comment installer le système Caml Light sur votre machine et fournit un index des mots-clés et des fonctions de bibliothèque, ainsi qu'une bibliographie sur les langages de la famille ML.

Le système Caml Light

Le système Caml Light dont il est question dans les parties II et III de cet ouvrage est un environnement de programmation en Caml fonctionnant à la fois sur stations de travail Unix et sur micro-ordinateurs Macintosh et PC (des portages sont également disponibles pour Atari ST et Amiga).

Le système Caml Light est distribué gratuitement et peut être reproduit librement à des fins non commerciales. On peut l'obtenir par l'intermédiaire du réseau Internet, en faisant FTP sur la machine `ftp.inria.fr`, dans le répertoire `lang/caml-light`. L'Institut National de Recherche en Informatique et en Automatique (INRIA) en assure également la distribution sur disquettes pour les micro-ordinateurs Macintosh et compatibles PC. Pour obtenir une disquette pour l'une ou l'autre de ces machines, reportez-vous à l'encadré qui figure en page de copyright.

Conventions

Le système Caml Light se présente sous différentes versions : pour machine Unix, pour Macintosh et pour PC. Certaines parties de ce livre sont spécifiques à une version particulière du système Caml Light. Ces parties sont présentées ainsi :

Unix : Ceci concerne uniquement la version Unix.

Mac : Ceci concerne uniquement la version Macintosh.

86 : Ceci concerne uniquement la version PC 8086 (et 80286).

386 : Ceci concerne uniquement la version PC 80386.

PC : Ceci concerne uniquement les deux versions PC.

I

Manuel de référence du langage Caml

1
Définition du langage Caml

Introduction

Ce chapitre constitue le manuel de référence du langage Caml. On y passe en revue toutes les constructions du langage et on y décrit leur syntaxe et leur sémantique. Ce n'est absolument pas une introduction au langage Caml, mais un texte de référence pour le programmeur Caml averti. On y suppose une solide connaissance préalable du langage, correspondant à la lecture de l'ouvrage *Le langage Caml*.

Ce chapitre définit formellement la syntaxe du langage Caml à l'aide de la notation BNF bien connue. Il explique également la sémantique des constructions, mais d'une manière informelle, sans prétendre à la rigueur mathématique : les mots sont employés dans leur sens intuitif, sans définition plus élaborée. Nous avons omis les règles de typage statique des constructions, afin d'éviter l'introduction de la théorie mathématique nécessaire à les exprimer. Nous faisons appel à l'intuition du lecteur pour retrouver les règles de typage à partir de la sémantique des constructions du langage.

Le lecteur curieux de savoir à quoi ressemble une description complètement formelle d'un langage de la famille ML peut s'attaquer à *The Definition of Standard ML* et à *Commentary on Standard ML*, par Milner, Tofte et Harper, MIT Press.

Le langage et ses implémentations

Dans ce chapitre, nous distinguons soigneusement le langage Caml, d'une part, et d'autre part les implémentations du langage Caml, comme le système Caml Light par exemple. La définition du langage Caml laisse volontairement un certain nombre de points non spécifiés, ou partiellement spécifiés, rendant les implémentations libres de choisir un comportement qui respecte les contraintes. Par exemple, dans le cas des expressions de paires `(`e_1`,`e_2`)`, il est dit explicitement que l'ordre d'évaluation de e_1 et e_2 n'est pas spécifié. Cela signifie qu'une implémentation de Caml peut choisir d'évaluer toujours e_1 d'abord, ou toujours e_2 d'abord, ou encore varier l'ordre d'évaluation à sa guise. Un programme qui fait l'hypothèse que e_2 est évaluée d'abord, comme c'est le cas dans la version courante du système Caml Light, dépend de l'implémentation et peut très bien donner des résultats différents s'il est exécuté par une autre implémentation. De même, les implémentations peuvent offrir des directives et des constructions syntaxiques en plus de celles qui sont décrites ici ; mais ces extensions ne font pas par-

tie du langage Caml et peuvent changer ou disparaître lors du passage à une autre implémentation, alors que les constructions décrites dans ce chapitre sont garanties dans toutes les implémentations de Caml.

Notations

La syntaxe du langage est décrite dans une notation proche de la BNF. Les symboles terminaux sont composés en police « machine à écrire » (`comme ceci`). Ils représentent des suites de caractères qui apparaissent tels quels dans le texte du programme. Les symboles non terminaux sont écrits en italiques (*comme cela*). Ils sont définis par des règles de production de la forme :

nonterminal ::= `hello`
| `(` *nonterminal* `)`

Cette définition se lit : « les objets de la classe *nonterminal* sont la suite de caractères `hello`, ou bien une parenthèse ouvrante suivie d'un objet de la classe *nonterminal* et d'une parenthèse fermante ». À l'intérieur des règles de production, on emploie les notations suivantes :

Notation	Signification	Exemple
[ ]	partie optionnelle	`let` [`rec`] signifie `let` ou `let rec`.
|	alternative	`0` | `1` signifie `0` ou `1`.
...	intervalle	`0`...`9` signifie un chiffre.
{ }	répétition	{`a`} signifie zéro, un ou plusieurs `a`.
$\{\ \}^+$	répétition stricte	{`0`...`9`}$^+$ signifie un ou plusieurs chiffres.
( )	regroupement	(`0` | `1`) `2` signifie `0` ou `1`, puis `2`.

1.1 Conventions lexicales

1.1.1 Identificateurs

ident ::= *lettre* {*lettre* | `0`...`9` | `_` | `'`}

lettre ::= `A`...`Z` | `a`...`z`

Les identificateurs sont des suites de lettres, chiffres, `_` (le caractère souligné) et `'` (le caractère apostrophe), qui commencent par une lettre. Les lettres contiennent au moins les 52 lettres majuscules et minuscules du jeu de caractères ASCII. L'implémentation peut autoriser parmi les lettres des caractères hors du jeu de caractères ASCII, c'est-à-dire de code supérieur ou égal à 128, comme par exemple les lettres accentuées dans le jeu ISO 8859–1. Les identificateurs ne doivent pas contenir deux caractères soulignés successifs (`__`). L'implémentation est susceptible de limiter le nombre de caractères d'un identificateur, mais cette limite doit être supérieure à 256 caractères. Tous les caractères d'un identificateur sont significatifs.

Ex : Les identificateurs suivants sont valides : `x`, `Y`, `toto`, `v12`, `x'1`, `valeur_x`.

Les identificateurs suivants sont invalides : `3x`, `_x`, `'x` (ne commencent pas par une lettre), `valeur__de__x` (contient la séquence `__`).

1.1.2 Entiers littéraux

entier-littéral	::=	[`-`] {`0`...`9`}$^+$
	|	[`-`] (`0x` | `0X`) {`0`...`9` | `A`...`F` | `a`...`f`}$^+$
	|	[`-`] (`0o` | `0O`) {`0`...`7`}$^+$
	|	[`-`] (`0b` | `0B`) {`0`...`1`}$^+$

Un entier littéral, aussi appelé « constante entière », est une suite d'un ou plusieurs chiffres, optionnellement précédée d'un signe moins. Par défaut, les entiers littéraux sont lus en base 10. Les préfixes suivants permettent de changer la base :

`0x`, `0X`	hexadécimal (base 16)
`0o`, `0O`	octal (base 8)
`0b`, `0B`	binaire (base 2).

(Le `0` initial est le chiffre zéro ; le `O` des constantes octales est la lettre O.) La signification des entiers littéraux qui sortent de l'intervalle des nombres entiers représentables est non spécifiée. L'intervalle des nombres entiers représentables contient au moins les entiers de -2^{30} à $2^{30}-1$, c'est-à-dire de -1073741824 à 1073741823. L'implémentation peut autoriser un intervalle plus grand de nombres entiers.

Ex : `123` représente l'entier 123.

`-456` représente l'entier -456.

`0xAB4C`, `0xab4c`, `0XAb4C` représentent l'entier 43852.

`-0o207` représente l'entier -135.

`0b101110001` représente l'entier 369.

La signification de `0x80000000` dépend de l'implémentation (ce nombre est plus grand que $2^{30}-1$).

1.1.3 Décimaux littéraux

décimal-littéral ::= [`-`] {`0`...`9`}$^+$ [`.` {`0`...`9`}] [(`e` | `E`) [`+` | `-`] {`0`...`9`}$^+$]

Les décimaux en notation flottante se composent d'une partie entière, une partie décimale et un exposant. La partie entière est une suite d'un ou plusieurs chiffres optionnellement précédée d'un signe moins. La partie décimale consiste en un point décimal suivi de zéro, un ou plusieurs chiffres. La partie exposant est formée d'un caractère `e` ou `E` optionnellement suivi d'un signe (`+` ou `-`), suivi d'un ou plusieurs chiffres. La partie décimale ou l'exposant peuvent être omis, mais pas les deux, afin

d'éviter l'ambiguïté avec les entiers littéraux. L'interprétation des décimaux littéraux dépend entièrement de l'implémentation, en particulier pour ce qui est de l'intervalle des valeurs représentables et de la précision de la représentation.

Ex : `1.2`, `-0.5`, `1e-5`, `-1.23E+10` sont des décimaux littéraux bien formés.

`1.` est un décimal littéral, mais `1` est un entier littéral.

`.5` n'est pas un décimal littéral.

1.1.4 Caractères littéraux

caractère-littéral ::= ` *caractère-normal* `
| ` \ (\ | ` | n | t | b | r) `
| ` \ (0...9) (0...9) (0...9) `

Les caractères littéraux sont encadrés par deux caractères ` (accent grave, aussi appelé *backquote*). Les deux accents graves délimitent soit un caractère différent de ` et \, soit l'une des séquences d'échappement suivantes :

Séquence	Caractère dénoté
`\\`	barre oblique (\, *backslash*)
`` \` ``	accent grave (`, *backquote*)
`\n`	saut de ligne (LF, *line feed*)
`\r`	retour chariot (CR, *carriage return*)
`\t`	tabulation horizontale (TAB)
`\b`	retour arrière (BS, *backspace*)
ddd	le caractère dont le code ASCII est *ddd* en décimal

Ex : `` `a` `` représente le caractère `a`.

`` `\`` `` représente le caractère ` (accent grave).

`` `\n` `` est le caractère « saut de ligne ».

`` `\098` `` est le caractère `b`.

`` `ab` ``, ` ``` `, `` `\7` ``, `` `\98` ``, `` `\1234` `` ne sont pas des caractères littéraux valides.

1.1.5 Chaînes de caractères littérales

chaîne-littérale ::= " {*caractère-de-chaîne*} "

caractère-de-chaîne ::= *caractère-normal*
| \ (\ | " | n | t | b | r)
| \ (0...9) (0...9) (0...9)

Les chaînes de caractères littérales sont encadrées par des caractères " (guillemets doubles). Les deux guillemets délimitent une suite de caractères différents de " et de \, ou de séquences d'échappement.

Séquence	Caractère dénoté
`\\`	barre oblique (\, *backslash*)
`\"`	guillemet (")
`\n`	saut de ligne (LF, *line feed*)
`\r`	retour chariot (CR, *carriage return*)
`\t`	tabulation horizontale (TAB)
`\b`	retour arrière (BS, *backspace*)
ddd	le caractère dont le code ASCII est *ddd* en décimal

L'implémentation doit autoriser des chaînes de caractères littérales d'une longueur au moins égale à $2^{16} - 1$ caractères (65535 caractères). L'implémentation peut autoriser des chaînes littérales plus longues.

Ex : `"Bonjour!"` représente la chaîne `Bonjour!`.

`"\r\n"` représente la chaîne « retour-chariot, saut de ligne ».

`"a\097c"` représente la chaîne `abc`.

`"\0974"` représente la chaîne `b4`.

1.1.6 Mots-clés

Les identificateurs suivants sont des mots-clés réservés du langage et ne peuvent être employés comme des identificateurs :

```
and   as    begin      do     done     downto
else  end   exception  for    fun      function
if    in    let        match  mutable  not
of    or    prefix     rec    then     to
try   type  value      where  while    with
```

Les caractères ou suites de caractères suivants sont aussi des mots-clés :

```
#   !   !=  &   (   )   *   *.  +   +.
,   -   -.  ->  .   .(  /   /.  :   ::
:=  ;   ;;  <   <.  <-  <=  <=. <>  <>.
=   =.  ==  >   >.  >=  >=. @   [   [|
]   ^   _   __  {   |   |]  }   '
```

1.1.7 Blancs

Les caractères suivants sont considérés comme des blancs : espace, retour-chariot, saut de ligne, saut de page, tabulation, tabulation verticale. Les blancs ne sont pas significatifs, sauf pour séparer deux identificateurs, deux littéraux ou deux mots-clés consécutifs, qui seraient sinon pris pour un seul identificateur, littéral ou mot-clé.

Ex : Les trois formes `1+x`, `1+␣␣x` et `1␣+␣x` sont lues de la même manière. (On a écrit ␣ pour marquer un espace.)

`f12` représente l'identificateur de nom `f12`.

`f␣12` est l'identificateur de nom `f`, suivi de l'entier littéral `12`.

1.1.8 Commentaires

Les commentaires commencent par les deux caractères `(*` et se terminent par les deux caractères `*)`. Il ne doit pas y avoir de blancs entre `(` et `*`, ni entre `*` et `)`. Les commentaires sont traités comme des blancs : ils ne sont pas significatifs, mais séparent les identificateurs, littéraux ou mots-clés consécutifs. Les séquences `(*` et `*)` ne sont pas reconnues comme débuts ou fins de commentaires lorsqu'elles apparaissent à l'intérieur de chaînes de caractères littérales. Les commentaires emboîtés (un commentaire à l'intérieur d'un autre) sont acceptés et correctement traités.

Ex : `1+(*commentaire*)2` est lu comme `1+2`.

`f(*commentaire*)x` est lu comme `f x`.

`1+(*commen(*tai*)re*)2` est lu comme `1+2`. Le premier `*)` ne termine pas le commentaire introduit par le premier `(*`.

Aucun texte n'est mis en commentaire dans `f "(*" x "*)"`, car les deux séquences `(*` et `*)` sont à l'intérieur de chaînes littérales.

`1+(*commen"*)"taire*)2` est lu comme `1+2`. Le `*)` entre guillemets ne termine pas le commentaire, puisqu'il est à l'intérieur d'une chaîne littérale.

1.1.9 Ambiguïtés

Les ambiguïtés lexicales sont résolues en suivant la règle du plus long préfixe ; quand une suite de caractères peut être décomposée en deux entités lexicales (encore appelées lexèmes ou tokens) de plusieurs manières différentes, la décomposition retenue est celle dont la première entité est la plus longue. On peut ajouter des blancs pour forcer une autre décomposition.

Ex : `f123.4` est lu comme l'identificateur `f123` suivi du mot-clé `.` et de l'entier littéral 4, et non pas comme l'identificateur `f` suivi du flottant littéral `123.4`.

`f 123.4` est lu comme l'identificateur `f` suivi du flottant littéral `123.4`.

1.2 Les noms globaux

Les noms globaux sont utilisés pour désigner quatre familles d'objets du langage :

- des valeurs liées à un nom global (aussi appelées valeurs globales),
- des constructeurs de valeurs (constants et non constants),
- des constructeurs de types,
- des noms de champs d'enregistrement (aussi appelés étiquettes).

Un nom global se compose de deux parties : le nom du module qui le définit et le nom local à l'intérieur de ce module. Les deux parties d'un nom global doivent être des identificateurs légaux (une lettre suivie de lettres, chiffres, soulignés et apostrophes).

1.2.1 Syntaxe des noms globaux

Le langage fournit deux syntaxes pour faire référence à un nom global :

nom-global	::=	*ident*
	|	$ident_1$ `__` $ident_2$

La forme $ident_1$ `__` $ident_2$ est appelée « nom complet » ou encore « nom qualifié ». Le premier identificateur est le nom du module définissant le nom global, le second est le nom local à l'intérieur de ce module. La version *ident* est appelée « nom abrégé » ou encore « nom non qualifié ». L'identificateur spécifie la partie « nom local » du nom global. Le nom du module définissant ce global est omis. Le compilateur le déduit du contexte en appliquant les règles de complétion du paragraphe suivant (1.2.2).

Ex : `list__map` représente le nom global défini dans le module `list` sous le nom local `map`.

`sub_string` représente un nom global dont le nom local est `sub_string` et dont le module de définition va être déterminé par le compilateur en suivant les règles de complétion.

1.2.2 Complétion des noms abrégés

Pour compléter un nom abrégé, le compilateur examine une liste de noms de modules, appelés *modules ouverts*, pour déterminer si l'un d'entre eux définit un nom global dont la partie « nom local » est égale au nom abrégé. Si c'est le cas, il complète le nom abrégé par le nom du module ouvert qui le définit. Si aucun module ouvert ne convient, le compilateur signale une erreur. Si plusieurs modules ouverts conviennent, le compilateur choisit celui qui apparaît en premier dans la liste des modules ouverts.

La liste des modules ouverts contient toujours en tête le nom du module en cours de compilation, qui est donc toujours examiné le premier. (Dans le cas d'une implémentation à base de boucle interactive, c'est le module qui contient toutes les définitions entrées au clavier.) Elle comprend aussi un certain nombre de modules de bibliothèque (examinés en dernier) qui sont fournis par l'environnement initial. Enfin, les directives `#open` et `#close` permettent d'ajouter ou de supprimer des noms de modules de cette liste. Les noms de modules ajoutés par `#open` sont examinés après le module en cours de compilation, mais avant les modules de la bibliothèque standard.

Ex : On se place dans la situation suivante. Le module en cours de compilation s'appelle `top` et contient une seule définition, celle du nom `f`. Les modules ouverts sont, en plus de `top`, les modules `list` et `string`. Le module `list` définit les noms `map` et `length`. Le module `string` définit les noms `length` et `sub_string`.

Le nom abrégé `f` est complété en `top__f`.

Le nom abrégé `sub_string` est complété en `string__sub_string`.

Le nom abrégé `length` est complété en `list__length`, si `list` précède `string` dans la liste des modules ouverts, et en `string__length` sinon.

Le nom abrégé `g` ne peut être complété et produit une erreur.

1.2.3 Identificateur de valeurs, de constructeurs, d'étiquettes

```
          valeur-globale ::= nom-global
                           | prefix nom-d'opérateur

         nom-d'opérateur ::= + | - | * | / | mod | +. | -. | *. | /.
                           | @ | ^ | ! | := | = | <> | == | != | !
                           | < | <= | > | <= | <. | <=. | >. | <=.

    constructeur-constant ::= nom-global
                           | []
                           | ()

constructeur-non-constant ::= nom-global
                           | prefix ::

     constructeur-de-type ::= nom-global

               étiquette ::= nom-global
```

Selon le contexte, les noms globaux désignent des valeurs globales, des constructeurs de valeurs constants, des constructeurs de valeurs non constants, des constructeurs de type, ou des noms de champs d'enregistrement, aussi appelés étiquettes. Pour les noms de valeurs et les constructeurs de valeurs, des noms spéciaux construits avec le mot-clé `prefix` et un nom d'opérateur sont aussi admis. Les lexèmes `[]` et `()` sont reconnus comme des constructeurs constants prédéfinis (la liste vide et la valeur « rien »).

Il n'y a pas de distinction syntaxique entre les noms globaux représentant des valeurs, des constructeurs et des étiquettes. Dans le cas des étiquettes et des constructeurs de types, la syntaxe du langage les autorise uniquement dans des contextes où aucune autre espèce de noms globaux ne peut apparaître. Il n'y a donc pas d'ambiguïté à donner le même nom global à une étiquette, un constructeur de type et une valeur globale (ou un constructeur de valeurs). Autrement dit, le langage fournit un espace de noms pour les étiquettes, un autre espace de noms pour les constructeurs de types et un troisième espace de noms pour les valeurs et constructeurs de valeurs.

Les constructeurs de valeurs et les valeurs globales partagent le même espace de noms. Un nom global en position de valeur est interprété comme un constructeur de valeurs s'il apparaît dans la portée d'une définition de type qui définit ce constructeur ; sinon, le nom global est interprété comme une valeur globale. Dans le cas d'un constructeur de valeurs, la définition de type qui l'introduit détermine si le constructeur est constant ou non.

1.3 Valeurs

Cette section décrit les valeurs manipulées par les programmes Caml Light.

1.3.1 Nombres entiers

Les valeurs entières sont les nombres entiers dans l'intervalle $[-2^{30}, 2^{30} - 1]$, c'est-à-dire de -1073741824 à 1073741823. L'implémentation peut autoriser un intervalle de

valeurs entières plus grand.

1.3.2 Nombres décimaux

Les valeurs décimales sont des nombres décimaux représentés en virgule flottante, c'est-à-dire sous la forme $m.2^e$ où la mantisse m et l'exposant e sont deux nombres entiers dans des intervalles non spécifiés. Toutes les caractéristiques des valeurs décimales dépendent de l'implémentation et de la machine, en particulier l'intervalle des nombres représentables, le nombre de chiffres significatifs et la façon dont les résultats sont arrondis. On encourage les implémentations à utiliser si possible des flottants « double précision » à la norme IEEE.

1.3.3 Caractères

Les caractères sont représentés comme des entiers entre 0 et 255. Les codes des caractères entre 0 et 127 sont interprétés d'après le standard ASCII. L'interprétation des codes des caractères entre 128 et 255 dépend de l'implémentation et de la machine.

1.3.4 Chaînes de caractères

Les chaînes de caractères sont des suites finies de caractères. L'implémentation doit autoriser des chaînes jusqu'à $2^{16} - 1$ caractères de longueur (65535 caractères). L'implémentation peut autoriser des chaînes plus longues.

1.3.5 N-uplets

Les n-uplets de valeurs sont des suites finies de valeurs. Le n-uplet des valeurs $v_1, \ldots, v_n$ est noté $(v_1,\ \ldots,\ v_n)$. L'implémentation doit autoriser les n-uplets ayant jusqu'à $2^{14}-1$ éléments (16383 éléments) et peut autoriser des n-uplets ayant davantage d'éléments.

1.3.6 Enregistrements

Les valeurs enregistrements sont des n-uplets de valeurs à champs nommés par des étiquettes. L'enregistrement noté $\{\mathit{étiquette}_1 = v_1;\ \ldots;\ \mathit{étiquette}_n = v_n\}$ associe la valeur v_i à l'étiquette $\mathit{étiquette}_i$, pour $i = 1,\ \ldots,\ n$. L'implémentation doit autoriser les enregistrements ayant jusqu'à $2^{14} - 1$ champs (16383 champs) et peut autoriser des enregistrements ayant davantage de champs.

1.3.7 Tableaux

Les tableaux sont des suites finies, de taille variable, de valeurs du même type. L'implémentation doit autoriser les tableaux ayant jusqu'à $2^{14} - 1$ éléments (16383 éléments) et peut autoriser des tableaux plus grands.

1.3.8 Valeurs de types somme

Une valeur d'un type somme est soit un constructeur constant, soit une paire d'un constructeur non constant et d'une valeur. Le premier cas est noté *constructeur-constant*, le second est noté *constructeur-non-constant*(v), où la valeur v est appelée argument du constructeur.

Les constantes suivantes sont des constructeurs constants prédéfinis :

`false`	le booléen « faux »	`()`	la valeur « rien »
`true`	le booléen « vrai »	`[]`	la liste vide

Le constructeur de listes `::` est un constructeur non constant prédéfini. Les listes sont des cas particuliers de valeurs de types somme : elles sont engendrées par les constructeurs `[]` et `::`. Le constructeur `[]` représente la liste vide. La liste dont la tête est v_1 et le reste v_2 est représentée par `::`(v_1, v_2), c'est-à-dire le constructeur `::` avec pour argument la paire (v_1, v_2).

1.3.9 Fonctions

Les valeurs fonctionnelles sont des fonctions partielles des valeurs dans les valeurs.

1.4 Expressions de type

```
exprtype ::= ' ident
           | ( exprtype )
           | exprtype -> exprtype
           | exprtype {* exprtype}+
           | constructeur-de-type
           | exprtype constructeur-de-type
           | ( exprtype {, exprtype} ) constructeur-de-type
```

La table ci-dessous indique les priorités relatives et l'associativité des opérateurs et des constructions de type qui ne sont pas fermées. Les constructions qui ont la plus grande priorité viennent en tête.

Opérateur	Associativité
Constructeur de type	–
`*`	–
`->`	à droite

Les expressions de type dénotent des types dans les définitions de types de données et dans les contraintes de type sur les motifs et les expressions.

Variables de type

L'expression de type `'` *ident* dénote la variable de type nommée *ident*. Dans les définitions de types de données, les variables de type sont des noms pour les paramètres du type de données. Dans les contraintes de type, elles représentent des types non spécifiés qui peuvent être remplacés par un type quelconque de manière à satisfaire la contrainte de type.

Types parenthésés

L'expression de type `(` *exprtype* `)` dénote le même type que *exprtype*.

Types fonctionnels

L'expression de type $exprtype_1$ `->` $exprtype_2$ dénote le type des fonctions qui font correspondre des résultats de type $exprtype_2$ à des arguments de type $exprtype_1$.

Types produits

L'expression de type $exprtype_1$ `*` $\cdots$ `*` $exprtype_n$ dénote le type des n-uplets dont les éléments appartiennent aux types $exprtype_1, \ldots, exprtype_n$ respectivement.

Types construits

Les constructeurs de type sans paramètre, comme *constructeur-de-type*, sont des expressions de type.

L'expression de type *exprtype constructeur-de-type*, où *constructeur-de-type* est un constructeur de type avec un paramètre, dénote l'application du constructeur *constructeur-de-type* au type dénoté par *exprtype*.

L'expression de type $(exprtype_1, \ldots, exprtype_n)$ *constructeur-de-type*, où *constructeur-de-type* est un constructeur de type avec n paramètres, dénote l'application du constructeur *constructeur-de-type* aux types $exprtype_1, \ldots, exprtype_n$.

Ex : On suppose que `int` et `string` sont des constructeurs de type sans arguments, correspondant aux types des valeurs entières et des chaînes de caractères ; `vect` un constructeur de type à un argument, correspondant au type des tableaux ; `hashtbl__t` un constructeur de type à deux arguments, correspondant au type des tables de hachage. Les expressions de types suivantes sont valides :

Expression	Type dénoté
`int`	les entiers
`int vect`	les tableaux d'entiers
`string * int`	les paires de chaînes et d'entiers
`(string,int) hashtbl__t`	les tables de hachage entre chaînes et entiers
`int -> int`	les fonctions des entiers dans les entiers
`int -> int * int`	les fonctions des entiers dans les paires d'entiers
`(int -> int) * int`	les paires formées d'une fonction des entiers dans les entiers et d'un entier
`int -> int vect`	les fonctions des entiers dans les tableaux d'entiers
`(int -> int) vect`	les tableaux de fonctions des entiers dans les entiers
`'a -> 'a`	les fonctions d'un certain type `'a` dans le même type `'a`

1.5 Constantes

constante ::= *entier-littéral*
| *décimal-littéral*
| *caractère-littéral*
| *chaîne-littérale*
| *constructeur-constant*

La classe syntaxique des constantes comprend les littéraux des quatre types de base (entiers, flottants, caractères, chaînes de caractères) et les constructeurs constants.

Ex : `-15`, `3.14`, `` `a` ``, `"mot"`, `[]` sont des constantes.

1.6 Motifs

motif ::= *ident*
| _
| *motif* `as` *ident*
| `(` *motif* `)`
| `(` *motif* `:` *exprtype* `)`
| *motif* `|` *motif*
| *constante*
| *constructeur-non-constant motif*
| *motif* `,` *motif* {`,` *motif*}
| `{` *étiquette* `=` *motif* {`;` *étiquette* `=` *motif*} `}`
| `[ ]`
| `[` *motif* {`;` *motif*} `]`
| *motif* `::` *motif*

La table ci-dessous indique les priorités relatives et l'associativité des opérateurs et des constructions de motifs qui ne sont pas fermées. Les constructions qui ont la plus grande priorité viennent en tête.

Opérateur	Associativité
Constructeur	–
`::`	à droite
`,`	–
`|`	à gauche
`as`	–

Les motifs sont des modèles qui permettent la sélection de structures de données ayant une certaine forme et la liaison de leurs composantes à des identificateurs. Cette opération de sélection est appelée filtrage (en anglais *pattern matching*). Son résultat est soit « cette valeur n'est pas filtrée par ce motif », soit « cette valeur est filtrée par ce motif et produit cet ensemble de liaisons identificateur-valeur ».

1.6.1 Motifs variables

Un motif réduit à un identificateur filtre n'importe quelle valeur, liant l'identificateur à la valeur. Le motif `_` filtre lui aussi n'importe quelle valeur, mais n'opère pas de liaison. Un identificateur ne doit pas apparaître plus d'une fois dans un motif donné.

Ex : Le motif `x` filtre la valeur 3 et lie `x` à 3.

Le motif `(x,x)` est incorrect, car il contient deux occurrences du même identificateur `x`.

1.6.2 Motifs parenthésés

Le motif `(` $motif_1$ `)` filtre les mêmes valeurs que $motif_1$. Une contrainte de type peut apparaître à l'intérieur d'un motif parenthésé, comme par exemple `(` $motif_1$ `:` *type* `)`. Cette contrainte oblige le type de $motif_1$ à être compatible avec *type*.

1.6.3 Motifs constants

Un motif réduit à une constante filtre les valeurs qui sont égales à cette constante.

Ex : Le motif `3` filtre la valeur entière 3 et aucune autre valeur.

Le motif `[]` filtre la liste vide et aucune autre valeur.

1.6.4 Motifs « ou »

Le motif *motif*$_1$ `|` *motif*$_2$ représente le « ou » logique des deux motifs *motif*$_1$ et *motif*$_2$. Une valeur est filtrée par *motif*$_1$ `|` *motif*$_2$ si elle est filtrée par *motif*$_1$ ou par *motif*$_2$. Les deux sous-motifs *motif*$_1$ et *motif*$_2$ ne doivent pas contenir d'identificateurs. Aucune liaison n'est donc réalisée lors d'un filtrage par un motif « ou ».

Ex : Le motif `1|2|3` filtre les valeurs entières 1, 2 et 3 uniquement. valeur.

Le motif `1|x` est incorrect, car il contient l'identificateur `x`.

1.6.5 Motifs « alias »

Le motif *motif*$_1$ `as` *ident* filtre les mêmes valeurs que *motif*$_1$. Si le filtrage par *motif*$_1$ réussit, l'identificateur *ident* est lié à la valeur filtrée, ce qui s'ajoute aux liaisons réalisées lors du filtrage par *motif*$_1$.

Ex : Le motif `1|2 as x` filtre les valeurs 1 et 2, tout en liant `x` à la valeur filtrée.

1.6.6 Motifs de types somme

Le motif *constructeur-non-constant* *motif*$_1$ filtre toutes les valeurs de type somme dont le constructeur est égal à *constructeur-non-constant* et dont l'argument est filtré par *motif*$_1$.

Le motif *motif*$_1$ `::` *motif*$_2$ filtre les listes non vides dont la tête filtre *motif*$_1$ et dont la queue filtre *motif*$_2$. Ce motif est équivalent à `prefix :: (` *motif*$_1$ `,` *motif*$_2$ `)`.

Le motif `[` *motif*$_1$ `;` ... `;` *motif*$_n$ `]` filtre les listes de longueur n dont les éléments filtrent *motif*$_1$ ... *motif*$_n$, respectivement. Ce motif est équivalent à *motif*$_1$ `::` ... `::` *motif*$_n$ `:: []`.

Ex : Si `C` est un constructeur non constant, le motif `C x` filtre les valeurs ayant `C` comme constructeur et lie `x` à la valeur argument de `C`.

Le motif `t::r` filtre les listes non vides, et lie `t` au premier élément de la liste et `r` au reste de la liste.

Le motif `[_; 0; _]` filtre les listes à trois éléments ayant la valeur 0 en deuxième position.

1.6.7 Motifs n-uplets

Le motif *motif*$_1$ `,` ... `,` *motif*$_n$ filtre les n-uplets dont les composantes filtrent les motifs *motif*$_1$ ... *motif*$_n$ respectivement. Autrement dit, ce motif filtre les n-uplets de valeurs $(v_1, \ldots, v_n)$ tels que *motif*$_i$ filtre v_i pour $i = 1, \ldots, n$.

Ex : Le motif `(1,x)` filtre toutes les paires dont le premier composant est l'entier 1 et lie `x` à la valeur du second composant.

1.6.8 Motifs enregistrements

Le motif `{` *étiquette*$_1$ `=` *motif*$_1$ `;` ... `;` *étiquette*$_n$ `=` *motif*$_n$ `}` filtre les enregistrements qui définissent au moins les champs *étiquette*$_1$ à *étiquette*$_n$, et tels que le motif *motif*$_i$ filtre la valeur associée à *étiquette*$_i$, pour $i = 1, \ldots, n$. La valeur enregistrement peut contenir d'autres champs en plus de *étiquette*$_1$ ... *étiquette*$_n$; les valeurs associées aux champs supplémentaires sont ignorées par le mécanisme de filtrage.

Ex : Le motif `{a=1; b=x}` filtre la valeur enregistrement associant 1 à l'étiquette `a`, 2 à `b` et 3 à `c`, et lie `x` à 2. Plus généralement, ce motif filtre tous les enregistrements associant 1 à `a` et une certaine valeur à `b`, et lie `x` à la valeur associée à `b`.

1.7 Expressions

```
expr            ::= ident
                  | valeur-globale
                  | constante
                  | ( expr )
                  | begin expr end
                  | ( expr : exprtype )
                  | expr , expr {, expr}
                  | constructeur-non-constant expr
                  | expr :: expr
                  | [ expr {; expr} ]
                  | [| expr {; expr} |]
                  | { étiquette = expr {; étiquette = expr} }
                  | expr expr
                  | opérateur-préfixe expr
                  | expr opérateur-infixe expr
                  | expr . étiquette
                  | expr . étiquette <- expr
                  | expr .( expr )
                  | expr .( expr ) <- expr
                  | expr & expr
                  | expr or expr
                  | if expr then expr [else expr]
                  | while expr do expr done
                  | for ident = expr (to | downto) expr do expr done
                  | expr ; expr
                  | match expr with filtrage
                  | function filtrage
                  | fun {motif}+ -> expr
                  | try expr with filtrage
                  | let [rec] liaison-let {and liaison-let} in expr

filtrage        ::= motif -> expr {| motif -> expr}

liaison-let     ::= motif = expr
                  | variable {motif}+ = expr

opérateur-préfixe ::= - | -. | !

opérateur-infixe  ::= + | - | * | / | mod | +. | -. | *. | /. | @ | ^ | ! | :=
                    | = | <> | == | != | < | <= | > | <= | <. | <=. | >. | <=.
```

La table suivante indique les priorités relatives et l'associativité des opérateurs et des constructions d'expressions qui ne sont pas fermées. Les constructions qui ont la plus grande priorité viennent en tête.

Construction ou opérateur	Associativité
`!`	–
`.` `.(`	–
application de fonction	à droite
application de constructeur	–
`-` `-.` (préfixe)	–
`mod`	à gauche
`*` `*.` `/` `/.`	à gauche
`+` `+.` `-` `-.`	à gauche
`::`	à droite
`@` `^`	à droite
comparaisons (`=` `==` `<` etc.)	à gauche
`not`	–
`&`	à gauche
`or`	à gauche
`,`	–
`<-` `:=`	à droite
`if`	–
`;`	à droite
`let` `match` `fun` `function` `try`	–

1.7.1 Constantes

Les expressions réduites à une constante s'évaluent en cette constante.

1.7.2 Noms

Les expressions réduites à un nom s'évaluent à la valeur liée à ce nom dans l'environnement d'évaluation courant. Le nom peut être un nom complet (*ident* `__` *ident*) ou un nom abrégé (*ident*). Un nom complet représente toujours une valeur globale. Un nom abrégé représente ou bien un identificateur lié localement par une construction `let`, `function`, `fun`, `match`, `try` ou `for`, ou bien une valeur globale dont le nom complet est obtenu en complétant le nom abrégé par la méthode décrite dans la section 1.2.

1.7.3 Expressions parenthésées

Les expressions `(` *expr* `)` et `begin` *expr* `end` ont la même valeur que *expr*. Les deux constructions sont sémantiquement équivalentes, mais l'usage est d'utiliser `begin`...`end` à l'intérieur des structures de contrôle :

```
if ... then begin ... ; ... end else begin ... ; ... end
```

et les parenthèses `(`...`)` dans les autres cas de regroupement d'expressions.

Une contrainte de type peut apparaître à l'intérieur d'une expression parenthésée, comme par exemple `(` *expr* `:` *type* `)`. Cette contrainte oblige le type de *expr* à être compatible avec *type*.

1.7.4 Définition de fonctions

Il y a deux formes syntaxiques de définition de fonctions. La première est introduite par le mot-clé `function` :

```
function motif1 -> expr1
       | ...
       | motifn -> exprn
```

Cette expression s'évalue en une valeur fonctionnelle définie par cas sur son argument. Lorsque cette fonction est appliquée à une valeur v, elle tente de filtrer cette valeur par chacun des motifs $motif_1$, ..., $motif_n$. Si la valeur v est filtrée par le motif $motif_i$, l'expression $expr_i$ associée à ce motif est évaluée et sa valeur est le résultat de l'application de la fonction. L'évaluation de $expr_i$ a lieu dans un environnement enrichi par les liaisons effectuées pendant le filtrage.

Si plusieurs motifs filtrent l'argument, celui qui apparaît en premier dans la définition de la fonction est sélectionné. Si aucun motif ne filtre l'argument, l'exception `Match_failure` est déclenchée.

L'autre forme syntaxique de définition de fonctions est introduite par le mot-clé `fun` :

`fun` $motif_1 \ldots motif_n$ `->` *expr*

Cette expression calcule la même valeur fonctionnelle que l'expression

`function` $motif_1$ `->` ... `->` `function` $motif_n$ `->` *expr*

Par conséquent, cette expression définit une fonction « curryfiée à n arguments », c'est-à-dire une fonction qui renvoie une fonction qui ... (n fois) qui renvoie la valeur de *expr* dans l'environnement enrichi par les filtrages des n arguments par les motifs $motif_1, \ldots, motif_n$.

Ex : L'expression `function (x,y) -> x` définit la fonction « première projection » sur les paires.

La fonction `function [] -> [] | tete::reste -> reste` renvoie la liste donnée en argument, privée de son premier élément. Si la liste est vide, on renvoie la liste vide.

La fonction `function tete::reste -> reste` renvoie la liste donnée en argument privée de son premier élément. Si la liste est vide, l'exception `Match_failure` se déclenche.

La fonction `function 1 -> "un" | 2 -> "deux" | n -> "beaucoup"` renvoie la chaîne `"un"` lorsqu'on l'applique à la valeur 1, `"deux"` lorsqu'on l'applique à 2, et `"beaucoup"` lorsqu'on l'applique à une valeur entière qui n'est ni 1 ni 2.

`fun x y -> (x+y)/2` définit la fonction « moyenne arithmétique » sous la forme d'une fonction curryfiée à deux arguments, c'est-à-dire une fonction qui prend une valeur v_x et renvoie une fonction qui à toute valeur v_y associe la moyenne de v_x et v_y.

1.7.5 Application de fonctions

L'application de fonctions est dénotée par simple juxtaposition de deux expressions. L'expression $expr_1$ $expr_2$ évalue les expressions $expr_1$ et $expr_2$. L'expression $expr_1$ doit s'évaluer en une valeur fonctionnelle, laquelle est alors appliquée à la valeur de $expr_2$. L'ordre dans lequel les expressions $expr_1$ et $expr_2$ sont évaluées n'est pas spécifié.

Ex: `(function x -> x+1) 2` s'évalue en l'entier 3.

`(fun x y -> (x+y)/2) 1` s'évalue en la fonction qui à tout entier v_y associe l'entier $(1 + v_y)/2$.

1.7.6 Liaisons locales

Les constructions `let` et `let rec` lient localement des identificateurs à des valeurs. La construction

`let` $motif_1$ `=` $expr_1$ `and`... `and` $motif_n$ `=` $expr_n$ `in` *expr*

évalue $expr_1, \ldots, expr_n$ dans un certain ordre non spécifié, puis filtre leurs valeurs par les motifs $motif_1, \ldots, motif_n$ respectivement. Si les filtrages réussissent, *expr* est évaluée dans un environnement enrichi par les liaisons réalisées pendant les filtrages, et la valeur de *expr* est renvoyée pour valeur de toute l'expression `let`. Si l'un des filtrages échoue, l'exception `Match_failure` est déclenchée.

Variante syntaxique : au lieu d'écrire

ident `=` `fun` $motif_1 \ldots motif_m$ `->` *expr*

dans une expression `let`, on peut écrire à la place

ident $motif_1 \ldots motif_m$ `=` *expr*.

Les deux formes lient *ident* à la fonction curryfiée à m arguments

`fun` $motif_1 \ldots motif_m$ `->` *expr*.

Ex: `let x=1 and y=2 in x+y` s'évalue en l'entier 3.

`let f x = x+1 in f(3)` s'évalue en l'entier 4.

`let [x] = [] in x` déclenche l'exception `Match_failure`.

Les définitions récursives d'identificateurs sont introduites par `let rec` :

`let rec` $motif_1$ `=` $expr_1$ `and`... `and` $motif_n$ `=` $expr_n$ `in` *expr*

La seule différence avec la construction `let` est que les liaisons variables-valeurs réalisées par le filtrage avec les motifs $motif_1$, ..., $motif_n$ sont considérées comme déjà effectives quand les expressions $expr_1$, ..., $expr_n$ sont évaluées. Les expressions $expr_1$, ..., $expr_n$ peuvent donc faire référence aux identificateurs liés par les motifs $motif_1$, ..., $motif_n$, et supposer qu'ils ont la même valeur que dans *expr*, le corps de la construction `let rec`.

Ex : `let rec f = function 0 -> 1 | x -> x*f(x-1) in` ... lie localement `f` à la fonction « factorielle » sur les entiers positifs.

`let rec f = function 0 -> 1 | x -> x*g(x) and g x = f(x-1)` lie `f` à la fonction « factorielle » et `g` à la fonction « factorielle de son argument moins 1 ».

Ce comportement des définitions récursives est garanti si les expressions $expr_1$ à $expr_n$ sont des définitions de fonctions (`fun`... ou `function`...) et les motifs $motif_1$, ..., $motif_n$ réduits à une variable, comme dans

`let rec` $ident_1$ `= function` ... `and` ... `and` $ident_n$ `= function` ... `in` *expr*

Cette construction définit les identificateurs $ident_1$, ..., $ident_n$ comme des fonctions mutuellement récursives locales à *expr*. Le comportement des autres formes de définitions `let rec` dépend de l'implémentation.

Ex : `let rec f = function 0 -> 1 | x -> x*g(x) and g x = f(x-1)` est garanti.

`let rec x = 1::x in` ... dépend de l'implémentation, car l'expression liée à `x` n'est pas une définition de fonction.

`let rec f = map g and g = function` ... dépend de l'implémentation, car l'expression liée à `f` n'est pas une définition de fonction.

`let rec (f,g) =` ... dépend de l'implémentation, car le motif n'est pas réduit à une variable.

`let rec (f: int -> int) = function x ->` ... dépend de l'implémentation, car le motif n'est pas réduit à une variable.

1.7.7 Séquence

L'expression $expr_1$ `;` $expr_2$ évalue $expr_1$ d'abord, puis $expr_2$, et retourne la valeur de $expr_2$.

Ex : `f(1); g(2); ()` applique la fonction `f` à l'argument 1, puis la fonction `g` à l'argument 2, et renvoie la constante `()`. Les résultats de `f` et de `g` sont ignorés.

1.7.8 Conditionnelle

L'expression `if` $expr_1$ `then` $expr_2$ `else` $expr_3$ s'évalue en la valeur de $expr_2$ si $expr_1$ s'évalue en la valeur booléenne `true`, et en la valeur de $expr_3$ si $expr_1$ s'évalue en la valeur booléenne `false`.

La partie `else` $expr_3$ est facultative, son omission équivaut à `else ()`.

Ex : `if x mod 2 = 0 then "pair" else "impair"` s'évalue en `"pair"` si la valeur de `x` est divisible par 2, en `"impair"` sinon.

`if x > 0 then f x` applique `f` à `x` si `x` est strictement positif et renvoie `()` sinon.

1.7.9 Conditionnelle par cas

L'expression

```
match expr with  motif1 -> expr1
               | ...
               | motifn -> exprn
```

filtre la valeur de *expr* par les motifs $motif_1$ à $motif_n$. Si le filtrage par $motif_i$ réussit, l'expression associée $expr_i$ est évaluée et sa valeur renvoyée comme valeur de l'expression `match` tout entière. L'évaluation de $expr_i$ a lieu dans un environnement enrichi par les liaisons effectuées pendant le filtrage. Si plusieurs motifs filtrent la valeur de *expr*, le motif qui apparaît le premier dans l'expression `match` est sélectionné. Si aucun motif ne filtre la valeur de *expr*, l'exception `Match_failure` est déclenchée. L'expression `match` ci-dessus se comporte exactement comme l'application de fonction

(`function` $motif_1$ `->` $expr_1$ `|` ... `|` $motif_n$ `->` $expr_n$) (*expr*)

Ex : `match x with t::r -> r | _ -> []` s'évalue en la liste `x` privée de son premier élément si `x` est non vide, ou bien la liste vide si `x` est vide.

`match x with t::r -> reste` s'évalue en la liste `x` privée de son premier élément si `x` est non vide, et déclenche l'exception `Match_failure` si `x` est vide.

1.7.10 Opérateurs booléens

L'expression $expr_1$ `&` $expr_2$ s'évalue en `true` si les deux expressions $expr_1$ et $expr_2$ s'évaluent en `true` ; sinon, elle s'évalue en `false`. L'évaluation est séquentielle : on évalue $expr_1$ en premier et $expr_2$ n'est pas évaluée si $expr_1$ est `false`. Autrement dit, $expr_1$ `&` $expr_2$ se comporte exactement comme `if` $expr_1$ `then` $expr_2$ `else false`.

L'expression $expr_1$ `or` $expr_2$ s'évalue en `true` si l'une des expressions $expr_1$ et $expr_2$ s'évalue en `true` ; sinon, elle s'évalue en `false`. L'évaluation est séquentielle : on évalue $expr_1$ en premier et $expr_2$ n'est pas évaluée si $expr_1$ est `true`. Autrement dit, $expr_1$ `or` $expr_2$ se comporte exactement comme `if` $expr_1$ `then true else` $expr_2$.

Ex : L'évaluation de `(x = 0) or (y/x >= 2)` ne produit jamais de division par zéro, quelle que soit la valeur de `x`.

1.7.11 Boucles

L'expression `while` $expr_1$ `do` $expr_2$ `done` évalue $expr_2$ de manière répétée tant que l'expression $expr_1$ s'évalue en `true`. La condition de boucle $expr_1$ est évaluée et testée au début de chaque itération. L'expression `while`...`done` tout entière a pour résultat la valeur `()`.

Ex :

```
let l = ref 0 and n = ref x in
while !n >= 1 do n := !n/2; l := !l+1 done; !l
```

renvoie le logarithme entier en base 2 de `x`.

L'expression `for` *ident* `=` $expr_1$ `to` $expr_2$ `do` $expr_3$ `done` évalue d'abord les expressions $expr_1$ et $expr_2$ (les bornes) en des valeurs entières n et p. Puis le corps de la boucle $expr_3$ est évaluée de façon répétée, dans un environnement où la valeur locale *ident* est tour à tour liée aux valeurs n, $n+1$, ..., $p-1$, p. Le corps de la boucle n'est jamais évalué si $n > p$. L'expression `for` tout entière s'évalue en la valeur `()`.

L'expression `for` *ident* `=` $expr_1$ `downto` $expr_2$ `do` $expr_3$ `done` s'évalue de la même manière, sauf que *ident* est successivement lié aux valeurs n, $n-1$, ..., $p+1$, p. Le corps de la boucle n'est jamais évalué si $n < p$.

Ex : `for i = 1 to n do print_int i; print_newline() done` imprime à l'écran tous les entiers de 1 à `n`, et n'imprime rien si `n` < 1.

1.7.12 Gestion d'exceptions

L'expression

`try` *expr* `with` $motif_1$ `->` $expr_1$
`|` ...
`|` $motif_n$ `->` $expr_n$

évalue l'expression *expr* et retourne sa valeur si l'évaluation de *expr* ne déclenche aucune exception. Si l'évaluation de *expr* déclenche une exception, la valeur exceptionnelle est filtrée par les motifs $motif_1$ à $motif_n$. Si le filtrage par $motif_i$ réussit, l'expression associée $expr_i$ est évaluée et sa valeur devient la valeur de l'expression `try`. L'évaluation de $expr_i$ a lieu dans un environnement enrichi par les liaisons effectuées pendant le filtrage. Si plusieurs motifs filtrent la valeur de *expr*, celui qui apparaît le premier dans l'expression `try` est sélectionné. Si aucun motif ne filtre la valeur de *expr*, la valeur exceptionnelle est relancée (déclenchée à nouveau) et donc « traverse » silencieusement la construction `try`.

Ex : `try f(x) with Division_by_zero -> 0` renvoie le résultat de `f(x)` si l'application de `f` à `x` ne déclenche pas d'exception. Si l'application de `f` à `x` déclenche l'exception `Division_by_zero`, l'expression `try` tout entière s'évalue en `0`. Si `f(x)` déclenche une exception autre que `Division_by_zero`, l'expression `try` déclenche la même exception.

```
try f(x)
with Failure s -> print_string s; raise(Failure s)
   | exc       -> print_string "Exception!"; raise exc
```

se comporte comme l'application `f(x)` (même valeur résultat, mêmes exceptions déclenchées), mais affiche un message si une exception se déclenche : l'argument de l'exception si l'exception est `Failure`, ou un message par défaut sinon.

1.7.13 Produits

L'expression $expr_1$ `, ... ,` $expr_n$ s'évalue en le n-uplet des valeurs des expressions $expr_1$ à $expr_n$. L'ordre d'évaluation des sous-expressions n'est pas spécifié.

Ex : `(1+1, 2+2)` s'évalue en la valeur $(2, 4)$.

1.7.14 Sommes

L'expression *constructeur-non-constant expr* s'évalue en la valeur de type somme dont le constructeur est *constructeur-non-constant* et dont l'argument est la valeur de *expr*.

Ex : Si `C` est un constructeur non constant, l'expression `C(1+2)` s'évalue en la valeur `C`(3).

Les listes disposent d'une syntaxe spéciale : l'expression $expr_1$ `::` $expr_2$ représente l'application du constructeur **`prefix`** `::` à `(` $expr_1$ `,` $expr_2$ `)`, et donc s'évalue en la liste dont la tête est la valeur de $expr_1$ et dont la queue est la valeur de $expr_2$. L'expression `[` $expr_1$ `;` ... `;` $expr_n$ `]` est équivalente à $expr_1$ `::` ... `::` $expr_n$ `::` `[]` et donc s'évalue en la liste dont les éléments sont les valeurs de $expr_1 \ldots expr_n$.

1.7.15 Enregistrements

L'expression `{` $étiquette_1$ `=` $expr_1$ `;` ... `;` $étiquette_n$ `=` $expr_n$ `}` s'évalue en l'enregistrement $\{étiquette_1 = v_1;\ \ldots\ ; étiquette_n = v_n\}$, où v_i est la valeur de $expr_i$ pour $i = 1, \ldots, n$. Les étiquettes $étiquette_1$ à $étiquette_n$ doivent toutes appartenir au même type enregistrement, c'est-à-dire avoir été introduites dans une même définition de type enregistrement. Toutes les étiquettes appartenant à ce type enregistrement doivent apparaître exactement une fois dans l'expression, bien qu'elles puissent apparaître dans n'importe quel ordre. L'ordre dans lequel les expressions $expr_1$, ..., $expr_n$ sont évaluées n'est pas spécifié.

Ex : Supposons que `a` et `b` soient les deux étiquettes d'un type enregistrement `t1` et que l'étiquette `c` appartienne à un autre type enregistrement `t2`.

L'expression `{a=1+2; b=true}` s'évalue en l'enregistrement {`a` = 3; `b` = `true`}.

L'expression `{b=true; a=1+2}` s'évalue en le même objet enregistrement.

L'expression `{a=1}` est incorrecte (l'étiquette `b` n'est pas définie).

L'expression `{a=1; b=true; a=2}` est incorrecte (l'étiquette `a` est définie deux fois).

L'expression `{a=1; b=true; c=‘0‘}` est incorrecte (l'étiquette `c` n'appartient pas au même type enregistrement que les deux autres).

L'expression $expr_1$ `.` *étiquette* évalue $expr_1$ en une valeur enregistrement et retourne la valeur associée à *étiquette* dans cette valeur enregistrement.

L'expression $expr_1$ `.` *étiquette* `<-` $expr_2$ évalue $expr_1$ en une valeur enregistrement, qui est alors modifiée physiquement en remplaçant la valeur associée à *étiquette* dans cet enregistrement par la valeur de $expr_2$. Cette opération n'est autorisée que si *étiquette* a été déclaré `mutable` dans la définition du type enregistrement. L'expression $expr_1$ `.` *étiquette* `<-` $expr_2$ s'évalue en la valeur rien `()`.

1.7.16 Tableaux

L'expression `[|` *expr*$_1$ `;`...`;` *expr*$_n$ `|]` s'évalue en un tableau à n cases, dont les éléments sont initialisés avec les valeurs de *expr*$_1$ à *expr*$_n$ respectivement. L'ordre dans lequel ces expressions sont évaluées n'est pas spécifié.

L'expression *expr*$_1$ `.(` *expr*$_2$ `)` est équivalente à l'application `vect_item` *expr*$_1$ *expr*$_2$. Dans l'environnement initial, l'identificateur `vect_item` est résolu en la fonction prédéfinie qui retourne la valeur de l'élément numéro *expr*$_2$ dans le tableau dénoté par *expr*$_1$. Le premier élément porte le numéro 0 et le dernier élément le numéro $n-1$, où n est la taille du tableau. L'exception `Invalid_argument` est déclenchée si l'on tente d'accéder hors des limites du tableau.

L'expression *expr*$_1$ `.(` *expr*$_2$ `) <-` *expr*$_3$ est équivalente à l'application `vect_assign` *expr*$_1$ *expr*$_2$ *expr*$_3$. Dans l'environnement initial, l'identificateur `vect_assign` est résolu en la fonction prédéfinie qui modifie physiquement le tableau dénoté par *expr*$_1$, remplaçant l'élément numéro *expr*$_2$ par la valeur de *expr*$_3$. L'exception `Invalid_argument` est déclenchée si l'on tente d'accéder hors des limites du tableau. La fonction prédéfinie retourne `()`. Donc, l'expression *expr*$_1$ `.(` *expr*$_2$ `) <-` *expr*$_3$ s'évalue en la valeur rien `()`.

Ce comportement de *expr*$_1$ `.(` *expr*$_2$ `)` et de *expr*$_1$ `.(` *expr*$_2$ `) <-` *expr*$_3$ peut changer si la signification des identificateurs `vect_item` et `vect_assign` est modifiée, soit par redéfinition soit par modification de la liste des modules ouverts. (Voir la discussion du paragraphe suivant.)

1.7.17 Opérateurs

Les opérateurs notés *opérateur-infixe* dans la grammaire des expressions peuvent apparaître en position infixe (entre deux expressions). Les opérateurs notés *opérateur-préfixe* dans la grammaire peuvent apparaître en position préfixe (devant une expression).

L'expression *opérateur-préfixe expr* est interprétée comme l'application *ident expr*, où *ident* est l'identificateur associé à l'opérateur *opérateur-préfixe* dans la table de la figure 1.1. De la même façon, *expr*$_1$ *opérateur-infixe* *expr*$_2$ est interprétée comme l'application *ident* *expr*$_1$ *expr*$_2$, où *ident* est l'identificateur associé à l'opérateur *opérateur-infixe* dans la table de la figure 1.1.

Ex : `-x` est interprétée comme `minus x`.

`x/y` est interprétée comme `(prefix /) x y`.

Les identificateurs associés aux opérateurs sont ensuite qualifiés suivant les règles de la section 1.7.2. Dans l'environnement initial, ils s'évaluent en des fonctions prédéfinies dont le comportement est décrit dans la table.

Ex : Dans l'environnement initial, le nom abrégé `minus` s'évalue en la fonction « opposé » sur les valeurs entières. L'expression `-x` s'évalue donc en l'opposé de `x`.

Le comportement de *opérateur-préfixe expr* et *expr*$_1$ *opérateur-infixe* *expr*$_2$ peut changer si la signification de l'identificateur associé à *opérateur-préfixe* ou *opérateur-infixe* est changée, soit par redéfinition des identificateurs, soit par modification de la liste des modules ouverts à l'aide des directives `#open` et `#close`.

Opérateur	Identificateur associé	Comportement dans l'environnement par défaut
`+`	`prefix +`	Addition entière.
`-` (infixe)	`prefix -`	Soustraction entière.
`-` (préfixe)	`minus`	Opposé entier.
`*`	`prefix *`	Multiplication entière.
`/`	`prefix /`	Division entière. Déclenche `Division_by_zero` si le second argument est nul. Le résultat n'est pas spécifié si un argument est négatif.
`mod`	`prefix mod`	Modulo entier. Déclenche `Division_by_zero` si le second argument est nul. Le résultat n'est pas spécifié si un argument est négatif.
`+.`	`prefix +.`	Addition flottante.
`-.` (infixe)	`prefix -.`	Soustraction flottante.
`-.` (préfixe)	`minus_float`	Opposé flottant.
`*.`	`prefix *.`	Multiplication flottante.
`/.`	`prefix /.`	Division flottante. Le résultat n'est pas spécifié si le second argument est nul.
`@`	`prefix @`	Concaténation des listes.
`^`	`prefix ^`	Concaténation des chaînes.
`!`	`prefix !`	Déréférencement (renvoie le contenu courant de la référence donnée).
`:=`	`prefix :=`	Affectation d'une référence.
`=`	`prefix =`	Test d'égalité structurelle.
`<>`	`prefix <>`	Test d'inégalité structurelle.
`==`	`prefix ==`	Test d'égalité physique.
`!=`	`prefix !=`	Test d'inégalité physique.
`<`	`prefix <`	Test « inférieur » sur les entiers.
`<=`	`prefix <=`	Test « inférieur ou égal » sur les entiers.
`>`	`prefix >`	Test « supérieur » sur les entiers.
`>=`	`prefix >=`	Test « supérieur ou égal » sur les entiers.
`<.`	`prefix <.`	Test « inférieur » sur les flottants.
`<=.`	`prefix <=.`	Test « inférieur ou égal » sur les flottants.
`>.`	`prefix >.`	Test « supérieur » sur les flottants.
`>=.`	`prefix >=.`	Test « supérieur ou égal » sur les flottants.

Le comportement des opérateurs `+`, `-`, `*`, `+.`, `-.`, `*.` ou `/.` n'est pas spécifié si le résultat tombe en dehors de l'intervalle des nombres représentables (entiers ou flottants selon le cas).

Figure 1.1: Signification des opérateurs

Ex : `let prefix + x y = if x < y then x else y in 1+2` s'évalue en l'entier 1, puisque dans l'expression `1+2` l'identificateur `prefix +` est lié à la fonction « minimum ».

1.8 Définitions de types et d'exceptions

Cette section décrit les constructions qui définissent des types et des exceptions.

1.8.1 Définitions de types

définition-de-type	::=	`type` *def-type* {`and` *def-type*}
def-type	::=	*params ident* `=` *def-constr* {`|` *def-constr*}
	|	*params ident* `=` `{` *def-étiquette* {`;` *def-étiquette*} `}`
	|	*params ident* `==` *exprtype*
	|	*params ident*
params	::=	*rien*
	|	`'` *ident*
	|	`(` `'` *ident* {`,` `'` *ident*} `)`
def-constr	::=	*ident*
	|	*ident* `of` *exprtype*
def-étiquette	::=	*ident* `:` *exprtype*
	|	`mutable` *ident* `:` *exprtype*

Les définitions de type lient les constructeurs de types à des types de données : des types sommes, des types enregistrements, des abréviations de types, ou des types de données abstraits.

Les définitions de type sont introduites par le mot-clé `type`. Elles consistent en une ou plusieurs définitions simples, éventuellement mutuellement récursives, séparées par le mot-clé `and`. Chaque définition simple définit un constructeur de type.

Une définition simple consiste en un identificateur, éventuellement précédé d'un ou de plusieurs paramètres de type, et suivi par la description d'un type de données. L'identificateur est le nom local du constructeur de type en cours de définition. (La partie « nom de module » de ce constructeur de type est le nom du module en cours de compilation.) Les éventuels paramètres de type sont soit une variable de type `'` *ident*, dans le cas d'un constructeur de types avec un paramètre, soit une liste de variables de types `(` `'` $ident_1, \ldots,$ `'` $ident_n$ `)`, dans le cas d'un constructeur de types avec plusieurs paramètres. Ces paramètres de type peuvent apparaître dans les expressions de type du membre droit de la définition.

Types somme

La définition de type *params ident* `=` $def\text{-}constr_1$ `|` ... `|` $def\text{-}constr_n$ définit un type somme. Les définition de constructeurs $def\text{-}constr_1$, ..., $def\text{-}constr_n$ décrivent les con-

structeurs associés au type somme. La définition de constructeur *ident* `of` *exprtype* déclare le nom local *ident* (du module en cours de compilation) en tant que constructeur non constant, dont l'argument est de type *exprtype*. La définition de constructeur *ident* déclare le nom local *ident* (du module en cours de compilation) en tant que constructeur constant.

Ex : `type direction = Nord | Sud | Est | Ouest` définit `direction` comme un type somme à quatre constructeurs constants.

`type 'a option = None | Some of 'a` définit `option` comme un type somme, paramétré par le type `'a`, comprenant un constructeur constant `None` et un constructeur non-constant `Some` dont l'argument a le type `'a`.

`type 'a liste = Vide | Cellule of 'a * 'a liste` définit un type somme récursif `liste` à un paramètre, isomorphe au type des listes.

Types enregistrement

La définition de type

params ident `= {` *def-étiquette*$_1$ `;` ... `;` *def-étiquette*$_n$ `}`

définit un type enregistrement. Les définitions d'étiquettes *def-étiquette*$_1$, ..., *def-étiquette*$_n$ décrivent les étiquettes associés au type enregistrement. La définition d'étiquette *ident* `:` *exprtype* déclare le nom local *ident* dans le module en cours de compilation comme une étiquette, dont l'argument a le type *exprtype*. La définition d'étiquette `mutable` *ident* `:` *exprtype* se comporte de façon analogue mais elle autorise de plus la modification physique de l'argument de l'étiquette.

Ex : `type variable = {nom:string; mutable valeur:int}` définit le type `variable` comme un type enregistrement à deux champs, un champ `nom` contenant une valeur de type `string` ne pouvant pas être modifiée en place, et un champ `valeur` contenant une valeur de type `int` pouvant être modifiée en place par la construction x`.valeur <-` n.

```
type variable = {nom:string; mutable valeur:valeur}
and valeur = Inconnue | Connue of int | Meme of variable
```

définit un type enregistrement `variable` et un type somme `valeur` de façon mutuellement récursive.

Abréviations de type

La définition de type *params ident* `==` *exprtype* définit le constructeur de type *ident* comme une abréviation pour l'expression de type *exprtype*.

Ex : Après la définition `type 'a endo == 'a -> 'a`, l'expression de type `int endo` est équivalente à `int -> int`.

Types abstraits

La définition de type *params ident* définit *ident* comme un type abstrait. Si elle apparaît dans l'interface d'un module, cette définition permet d'exporter un constructeur de type tout en cachant sa représentation dans l'implémentation du module.

1.8.2 Définitions d'exceptions

définition-d'exception ::= `exception` *def-constr* {`and` *def-constr*}

Les définitions d'exceptions ajoutent de nouveaux constructeurs au type somme prédéfini `exn` des valeurs exceptionnelles. Les constructeurs sont déclarés comme dans le cas d'une définition de type somme.

Ex : `exception Fini and Erreur of string` définit deux nouveaux constructeurs du type `exn`, un constructeur constant nommé `Fini` et un constructeur non constant nommé `Erreur`.

1.9 Directives

directive ::= `# open` *chaîne*
| `# close` *chaîne*
| `#` *ident chaîne*

Les directives contrôlent le comportement du compilateur. Elles s'appliquent à la suite de l'unité de compilation courante (le reste du fichier dans le cas d'une compilation non interactive, le reste de la session dans le cas d'une utilisation interactive).

Les deux directives `#open` et `#close` modifient la liste des modules ouverts, que le compilateur utilise pour compléter les identificateurs non qualifiés, comme décrit dans la section 1.2. La directive `#open` *chaîne* ajoute le module de nom la chaîne littérale *chaîne* à la liste des modules ouverts, en première position. La directive `#close` *chaîne* supprime de la liste des modules ouverts la première occurrence du module de nom *chaîne*.

L'implémentation peut fournir d'autres directives, pourvu qu'elles répondent à la syntaxe `#` *ident chaîne*, où *ident* est le nom de la directive et la chaîne littérale *chaîne* l'argument de la directive. Le comportement de ces directives supplémentaires dépend de l'implémentation.

1.10 Implémentations de modules

implémentation	::=	{*phrase-d'implémentation* `;;`}
phrase-d'implémentation	::=	*expr*
	|	*définition-de-valeur*
	|	*définition-de-type*
	|	*définition-d'exception*
	|	*directive*
définition-de-valeur	::=	`let` [`rec`] *liaison-let* {`and` *liaison-let*}

L'implémentation d'un module consiste en une suite de phrases d'implémentation, terminées par un double point-virgule. Une phrase d'implémentation est soit une expression, soit une définition de valeur, de type ou d'exception, soit une directive. Pendant l'exécution, les phrases d'implémentation sont évaluées en séquence, dans leur ordre d'apparition dans l'implémentation du module.

Les phrases d'implémentation réduites à une expression sont évaluées pour leurs effets.

Ex : La phrase `3;;` n'a pas d'effets observables à l'exécution.

La phrase `print_int(1+2); print_newline();;` affiche `3` sur la sortie standard.

Les définitions de valeur lient des variables de valeur globales de la même manière qu'une expression `let`...`in`... lie des variables locales. Les expressions sont évaluées et leurs valeurs sont filtrées par les parties gauches du signe `=` comme décrit dans la section 1.7.6. Si le filtrage réussit, les liaisons des identificateurs aux valeurs effectuées au cours du filtrage sont interprétées comme des liaisons de valeurs aux variables globales dont le nom local est l'identificateur, et dont le nom de module est le nom du module. Si le filtrage échoue, l'exception `Match_failure` est déclenchée. La portée de ces liaisons est l'ensemble des phrases qui suivent cette définition de valeur dans l'implémentation du module.

Ex : La phrase `let succ x = x+1;;` lie le nom global `succ` à la fonction « successeur » sur les entiers.

Les définitions de type et d'exception introduisent des constructeurs de type, des constructeurs de valeurs et des étiquettes d'enregistrement comme décrit dans les parties 1.8.1 et 1.8.2. La portée de ces définitions est l'ensemble des phrases qui suivent la définition dans l'implémentation du module. L'évaluation d'une phrase d'implémentation réduite à une définition de type ou d'exception ne produit aucun effet à l'exécution.

Les directives modifient le comportement du compilateur sur les phrases suivantes de l'implémentation du module, comme indiqué dans la section 1.9. L'évaluation d'une phrase d'implémentation réduite à une directive ne produit aucun effet à l'exécution. Les directives ne concernent que l'implémentation en cours de compilation ; en particulier, elles n'ont pas d'effet sur les autres modules qui font référence aux globaux exportés par l'interface en cours de compilation.

1.11 Interfaces de modules

interface	::=	{*phrase-d'interface* `;;`}
phrase-d'interface	::=	*déclaration-de-valeur*
	|	*déclaration-de-type*
	|	*déclaration-d-exception*
	|	*directive*
déclaration-de-valeur	::=	`value` *decl-val* {`and` *decl-val*}
decl-val	::=	*ident* `:` *expression-de-type*

Les interfaces de modules déclarent les objets globaux (valeurs globales, constructeurs de types, constructeurs de valeurs, étiquettes d'enregistrements) qu'un module exporte, c'est-à-dire rend accessibles aux autres modules. Les autres modules peuvent faire référence à ces globaux en utilisant des identificateurs qualifiés ou la directive `#open`, comme expliqué dans la section 1.2.

Une interface de module consiste en une suite de phrases d'interface, terminées par un double point-virgule. Une phrase d'interface est soit une déclaration de valeur, soit une déclaration de type, soit une déclaration d'exception, soit une directive.

Les déclarations de valeurs déclarent les valeurs globales qui sont exportées par l'implémentation du module et les types avec lesquels elles sont exportées. L'implémentation du module doit définir ces valeurs, avec des types au moins aussi généraux que les types déclarés dans l'interface. La portée des liaisons de ces valeurs globales s'étend de l'implémentation du module lui-même à tous les modules qui font référence à ces valeurs.

Ex : La phrase d'interface `value f: int -> int;;` indique que l'implémentation définit une valeur globale `f` de type `int -> int` et rend cette valeur accessible aux autres modules.

Les déclarations de type ou d'exception introduisent des constructeurs de types, des constructeurs de valeurs et des étiquettes d'enregistrements comme expliqué dans les sections 1.8.1 et 1.8.2. Les déclarations d'exception et de type qui ne sont pas abstraits prennent effet dans l'implémentation du module ; c'est-à-dire que les constructeur de types, les constructeurs de sommes et les étiquettes d'enregistrements qu'ils définissent sont considérés comme définis au début de l'implémentation du module, et les phrases d'implémentation peuvent y faire référence. Les déclarations de type qui ne sont pas abstraits ne doivent pas être redéfinies dans l'implémentation du module. Au contraire, les constructeurs de types qui ont été déclarés abstraits doivent être définis dans l'implémentation du module avec les mêmes noms.

Ex : La phrase d'interface `type nombre = Entier of int | Flottant of float;;` définit un type somme `nombre` et deux constructeurs `Entier` et `Flottant` qui sont utilisables à la fois dans l'implémentation du module et dans les autres modules.

La phrase d'interface `type nombre;;` définit un type abstrait `nombre`. L'implémentation du module doit contenir une définition de ce type, par exemple

`type nombre = N of int`. Le constructeur de type `nombre` est accessible aux autres modules, mais le constructeur de valeur `N` ne l'est pas.

Les directives modifient le comportement du compilateur sur les phrases suivantes de l'interface du module, comme il est indiqué dans la section 1.9. Les directives ne concernent que l'interface en cours de compilation ; en particulier, elles n'ont pas d'effet sur les autres modules qui « ouvrent » l'interface par la directive `#open`.

2
Extensions propres à Caml Light

Ce chapitre décrit des extensions au langage Caml qui sont implémentées dans le système Caml Light version 0.6, mais ne font pas partie du langage Caml de base. Comme tout ce qui n'est pas spécifié dans le chapitre 1, les extensions présentées ici sont sujettes à modification, voire suppression, dans le futur.

2.1 Flux, analyseurs syntaxiques et imprimeurs

Caml Light fournit un type prédéfini des flux (en anglais *streams*) : des suites de valeurs, éventuellement infinies, qui sont évaluées à la demande. Il fournit également des expressions et des motifs opérant sur les flux : les expressions de flux servent à construire des flux, tandis que les motifs de flux servent à accéder au contenu des flux. Les flux et motifs de flux sont particulièrement bien adaptés à l'écriture des analyseurs syntaxiques à descente récursive.

Les expressions de flux sont introduites par une extension de la classe syntaxique des expressions :

expr	::=	...
	|	`[< >]`
	|	`[<` *composant-de-flux* {`;` *composant-de-flux*} `>]`
	|	`function` *filtrage-de-flux*
	|	`match` *expr* `with` *filtrage-de-flux*
filtrage-de-flux	::=	*motif-de-flux* `->` *expr* {`|` *motif-de-flux* `->` *expr*}
composant-de-flux	::=	`'` *expr*
	|	*expr*

Les expressions de flux sont parenthésées par `[<` et `>]`. Elles représentent la concaténation de leurs composants. Le composant `'` *expr* représente le flux à un seul élément : la valeur de *expr*. Le composant *expr* représente un sous-flux. Le flux vide est dénoté par `[< >]`.

Ex : Si `s` et `t` sont des flux d'entiers, alors `[<'1; s; t; '2>]` est un flux d'entiers contenant l'élément `1`, puis les éléments de `s`, puis ceux de `t`, et finalement `2`.

Au contraire de toutes les autres expressions du langage, les expressions de flux sont soumises à l'évaluation paresseuse (en anglais *lazy evaluation*) : les composants d'un flux ne sont pas évalués à la construction du flux, mais seulement quand on accède à ces composants pendant le filtrage de flux. Les composants sont évalués une seule fois, lors du premier accès ; les accès suivants utilisent directement la valeur calculée la première fois.

Ex : L'expression `[< 1/0 >]` s'évalue sans déclencher l'exception `Division_by_zero`. L'exception ne se déclenche qu'au moment où on accède au premier élément de ce flux, lors d'un filtrage.

motif-de-flux ::= `[< >]`
| `[<` *composant* {`;` *composant*} `>]`

composant ::= `'` *motif*
| *expr motif*
| *ident*

Les motifs de flux, également parenthésés par `[<` et `>]`, décrivent des segments initiaux de flux : un motif de flux filtre tous les flux dont les premiers éléments sont de la forme spécifiée par le motif de flux, mais ces premiers éléments peuvent être suivis par un nombre arbitraire d'éléments quelconques. En particulier, le motif de flux `[< >]` filtre tous les flux. Le filtrage s'effectue séquentiellement sur chaque composant du motif de flux, de la gauche vers la droite. Chaque composant filtre un segment initial du flux, puis, si le filtrage réussit, retire du flux les éléments filtrés. Le composant suivant va donc filtrer les éléments suivants du flux. Le filtrage de flux opère destructivement : quand un élément est filtré, il est supprimé du flux par modification physique.

Le composant `'` *motif* filtre le premier élément du flux par le motif *motif*. Le composant *expr motif* applique la fonction dénotée par *expr* au flux courant, puis filtre par *motif* le résultat de la fonction. Finalement, le composant *ident* lie simplement l'identificateur *ident* au flux filtré. (L'implémentation actuelle restreint *ident* à figurer en dernier dans un motif de flux.)

Le filtrage de flux procède en deux étapes : d'abord, un motif est sélectionné par filtrage du flux par les premiers composants des différents motifs de flux ; puis, le reste du flux est filtré par les composants suivants du motif sélectionné. Si les composants suivants ne sont pas compatibles, l'exception `Parse_error` est déclenchée. Il n'y a pas de retour en arrière (en anglais *backtracking*) ici : le filtrage de flux choisit de manière irréversible le motif sélectionné par examen du premier composant. Si aucun des premiers composants des motifs de flux ne filtre, l'exception `Parse_failure` est déclenchée. Lorsque l'exception `Parse_failure` se déclenche pendant le filtrage avec la première composante d'un motif, le filtrage de flux l'intercepte et passe à la prochaine alternative.

Ex :
```
let oui_ou_non = function
      [< ' "oui" >] -> true
    | [< ' "non" >] -> false;;
let consent = function
```

```
    [< ' "bien"; ' "sûr" >] -> true
  | [< oui_ou_non x >] -> x
  | [< ' "euh"; oui_ou_non x >] -> x;;
```

Voici quelques exemples d'application de `consent` à divers flux :

`consent [< ' "bien"; ' "sûr"; ' "!" >]` a pour résultat `true`. Le premier cas de `consent` s'applique.

`consent [< ' "bien"; ' "entendu" >]` déclenche l'exception `Parse_error`. Le premier cas est sélectionné au vu du premier élément `bien`, mais le reste du flux ne filtre pas le reste du motif.

`consent [< ' "oui" >]` s'évalue en `true`. `oui_ou_non` est appliquée à ce flux et renvoie la valeur `true`, qui filtre `x`. Le deuxième cas est sélectionné, et réussit.

`consent [< ' "euh"; ' "non" >]` s'évalue en `false`. `oui_ou_non` déclenche `Parse_failure` sur ce flux, puisque `euh` ne filtre ni `oui` ni `non`. Le second cas est donc écarté, et on sélectionne le troisième cas au vu de `euh`. Le reste du filtrage réussit.

`consent [< ' "euh"; ' "eh bien" >]` déclenche `Parse_error`. Le troisième cas est sélectionné comme précédemment, mais le filtrage de `[< ' "eh bien" >]` par `oui_ou_non` échoue par déclenchement de `Parse_failure`, donc `consent` déclenche `Parse_error`.

`consent [< ' "une"; ' "minute" >]` déclenche `Parse_failure`. Aucun des cas de `consent` n'est sélectionné, puisque aucun des premiers composants ne filtre.

On se reportera à l'ouvrage *Le langage Caml*, chapitres 9 et suivants pour une introduction plus progressive aux flux, et pour des exemples de leur application à l'écriture d'analyseurs syntaxiques et lexicaux. Pour une présentation plus formelle des flux et une discussion d'autres sémantiques possibles, voir *Parsers in ML* par Michel Mauny et Daniel de Rauglaudre, dans les actes de la conférence ACM « Lisp and Functional Programming » 1992.

2.2 Motifs intervalles

Caml Light reconnaît les motifs de la forme `` ` ``c`` ` .. ` ``d`` ` `` (deux caractères littéraux séparées par deux points) comme une abréviation du motif

$$`c` \mid `c_1` \mid `c_2` \mid \ldots \mid `c_n` \mid `d`$$

où $c_1, c_2, \ldots, c_n$ sont les caractères qui se trouvent entre c et d dans l'ensemble des caractères ASCII. Par exemple, le motif `` `0`..`9` `` filtre tous les chiffres.

2.3 Définitions récursives de valeurs

En plus des définitions `let rec` de valeurs fonctionnelles, décrites au chapitre 1, Caml Light autorise une certaine classe de définitions récursives de valeurs non fonctionnelles. Par exemple, la définition `let rec x = 1 :: y and y = 2 :: x;;` est

acceptée ; son évaluation lie `x` à la liste cyclique `1::2::1::2::`... et `y` à la liste cyclique `2::1::2::1::`... Informellement, les définitions acceptées sont celles où les variables définies n'apparaissent qu'au sein du corps d'une fonction, ou bien constituent un champ d'une structure de données. De plus, les motifs des membres gauches doivent être des identificateurs ; rien de plus compliqué.

2.4 Types somme mutables

L'argument d'un constructeur de valeur peut être déclaré `mutable`, c'est-à-dire modifiable en place, lors de la définition du type :

```
type foo = A of mutable int
         | B of mutable int * int
         | ...
```

La modification en place de la partie argument d'une valeur s'effectue par la construction *ident* `<-` *expr*, où *ident* est un identificateur lié par filtrage de motif à l'argument d'un constructeur modifiable et *expr* dénote la valeur qui doit remplacer cet argument. En continuant l'exemple précédent :

```
let x = A 1 in
  begin match x with A y -> y <- 2 | _ -> () end; x
```

renvoie la valeur `A 2`. La notation *ident* `<-` *expr* est également acceptée si *ident* est un identificateur lié par filtrage à la valeur d'un champ modifiable d'enregistrement. Par exemple,

```
type bar = {mutable lbl : int};;
let x = {lbl = 1} in
  begin match x with {lbl = y} -> y <- 2 end; x
```

renvoie la valeur `{lbl = 2}`.

2.5 Directives

En plus des directives standard `#open` et `#close`, Caml Light fournit trois directives supplémentaires.

`#infix "` *id* `"`

Modifie le statut lexical de l'identificateur *id* : dans la suite de l'unité de compilation, *id* sera reconnu comme un opérateur infixe, au même titre que `+` par exemple. La notation `prefix` *id* permet de faire référence à l'identificateur *id* lui-même. Les expressions de la forme $expr_1$ *id* $expr_2$ sont analysées comme l'application `prefix` *id* $expr_1$ $expr_2$. L'argument de la directive `#infix` doit être un identificateur, c'est-à-dire une suite de lettres, chiffres et soulignés qui commence par une lettre ; sinon, la déclaration `#infix` n'a pas d'effet. Exemple :

```
#infix "union";;
let prefix union = fun x y -> ... ;;
[1,2] union [3,4];;
```

`#uninfix` " *id* "
: Supprime le statut d'infixe précédemment attaché à l'identificateur *id* par une directive `#infix` " *id* ".

`#directory` " *nom-de-répertoire* "
: Ajoute le répertoire donné au chemin d'accès des répertoires parcourus pour trouver les fichiers d'interface des modules. C'est équivalent à l'option `-I` de la ligne de commandes du compilateur indépendant et du système interactif.

II

Manuel d'utilisation du système Caml Light

3
Compilation indépendante (camlc)

Ce chapitre décrit comment les programmes Caml Light peuvent être compilés non interactivement, et transformés en fichiers exécutables. C'est le travail de la commande `camlc`, qui effectue la compilation et l'édition de liens (en anglais *linking*) des fichiers source Caml Light.

Mac : Cette commande n'est pas une application Macintosh indépendante. Pour exécuter `camlc`, vous devez posséder l'environnement de programmation Macintosh Programmer's Workshop (MPW). Les programmes engendrés par `camlc` sont des « outils » MPW, et non des applications Macintosh indépendantes.

3.1 Survol du compilateur

La commande `camlc` se contrôle *via* des paramètres sur la ligne de commande, dans le style de la plupart des compilateurs C. Elle accepte plusieurs types d'arguments : fichiers source pour les implémentations de modules, fichiers source pour les interfaces de modules, et implémentations compilées de modules.

- Les arguments se terminant par `.mli` sont considérés comme des fichiers source pour des interfaces de modules. Les interfaces de modules déclarent les identificateurs globaux exportés et définissent les types de données publics. À partir du fichier *x*`.mli`, le compilateur `camlc` produit une interface compilée dans le fichier *x*`.zi`.
- Les arguments se terminant par `.ml` sont considérés comme des fichiers source pour des implémentations de modules. Les implémentations de modules lient des identificateurs globaux à des valeurs, définissent des types de données privés et contiennent des expressions à évaluer pour leurs effets. À partir du fichier *x*`.ml`, le compilateur `camlc` produit du code compilé dans le fichier *x*`.zo`. Si le fichier d'interface *x*`.mli` existe, le système vérifie la conformité de l'implémentation du module *x*`.ml` avec l'interface correspondante *x*`.zi`, qui doit avoir été préalablement compilée. Si aucune interface *x*`.mli` n'est fournie, la compilation de *x*`.ml` produit un fichier d'interface compilée *x*`.zi` en plus du

fichier de code compilé x`.zo`. Le fichier x`.zi` obtenu correspond à une interface qui exporte tout ce qui est défini dans l'implémentation x`.ml`.

- Les arguments se terminant par `.zo` sont considérés comme des fichiers de code compilé. Ces fichiers sont liés entre eux, avec les fichiers de code obtenus en compilant les arguments `.ml` (s'il y en a), et avec la bibliothèque standard de Caml Light, pour produire un programme exécutable indépendant. L'ordre dans lequel les arguments `.zo` et `.ml` se présentent sur la ligne de commande est important : les identificateurs globaux sont initialisés dans cet ordre à l'exécution, et c'est une erreur d'utiliser un identificateur global avant de l'avoir initialisé (cette erreur est signalée au moment de l'édition de liens). Donc, un fichier x`.zo` donné doit apparaître avant tous les fichiers `.zo` qui font référence à l'un des identificateurs définis dans le fichier x`.zo`.

Le résultat de la phase d'édition de liens est un fichier contenant du code compilé qui peut être lancé par l'exécutant (en anglais *runtime system*) de Caml Light : la commande nommée `camlrun`. Si `caml.out` est le nom du fichier produit par la phase d'édition de liens, la commande

`camlrun caml.out` arg_1 arg_2 … arg_n

exécute le code compilé contenu dans `caml.out`, en lui passant pour arguments les chaînes de caractères arg_1 à arg_n. (Voir le chapitre 5 pour plus de détails.)

Unix : Sur la plupart des systèmes Unix, le fichier produit par l'édition de liens peut être exécuté directement, par exemple :

`./caml.out` arg_1 arg_2 … arg_n

Le fichier produit a le bit `x` et est capable de lancer lui-même l'exécutant.

PC : Le fichier produit par la phase d'édition de liens est directement exécutable, pourvu qu'il ait le suffixe `.EXE`. Donc, si l'on nomme le fichier produit `caml_out.exe`, vous pouvez le lancer par

`caml_out` arg_1 arg_2 … arg_n

En fait, le fichier produit `caml_out.exe` est formé d'un petit fichier exécutable précédant le code compilé. Cet exécutable lance simplement la commande `camlrun` sur le reste du fichier. (Ce qui fait que `caml_out.exe` n'est pas un exécutable indépendant : il a encore besoin que `camlrun.exe` réside dans l'un des répertoires du chemin d'accès.)

3.2 Options

Les options suivantes peuvent apparaître sur la ligne de commande.

`-c` Compilation seulement. Supprime la phase d'édition de liens. Les fichiers source produisent des fichiers compilés, mais il n'y a pas création de fichier exécutable. Cette option sert à compiler des modules séparément.

`-custom`

Édition de liens en mode « exécutant dédié » (en anglais *custom runtime*). Dans le mode par défaut, l'éditeur de liens produit du code exécutable par l'exécutant standard `camlrun`. Dans le mode « exécutant dédié », l'éditeur de liens produit un fichier qui contient à la fois le code d'un exécutant dédié et le code pour le programme. Le fichier résultant est considérablement plus gros, mais complètement indépendant cette fois. En particulier, il est directement exécutable, même si la commande `camlrun` n'est pas installée. De plus, le mode « exécutant dédié » permet de lier du code Caml Light avec des fonctions C définies par l'utilisateur, comme décrit au chapitre 8.

Unix : Ne jamais supprimer les symboles (par la commande `strip`) d'un exécutable produit avec l'option `-custom`.

PC : Cette option n'est pas implémentée.

Mac : Cette option n'est pas implémentée.

`-files` *fichier-de-réponses*

Traite les fichiers dont les noms sont listés dans le fichier *fichier-de-réponses* comme si ces noms apparaissaient sur la ligne de commande. Les noms de fichiers contenus dans *fichier-de-réponses* sont séparés par des blancs (espaces, tabulations ou sauts de ligne). Cette option permet de circonvenir de sottes limitations sur la longueur de la ligne de commande.

`-g` Fait produire des informations de mise au point (en anglais *debugging*) par le compilateur. Lors de la phase d'édition de liens, cette option ajoute des informations à la fin du fichier de code exécutable produit. Ces informations sont requises en particulier par le gestionnaire d'exception du module `printexc` de la bibliothèque standard (section 10.7).

Lors de la phase de compilation d'une implémentation (fichier `.ml`), l'option `-g` force le compilateur à produire un fichier avec l'extension `.zix` décrivant tous les types et les valeurs globales définies dans l'implémentations, y compris celles qui sont locales à l'implémentation, c'est-à-dire non décrites dans l'interface du module. En conjonction avec l'option `-g` du système interactif (chapitre 4), ce fichier `.zix` permet d'accéder aux valeurs locales d'un module à partir du système interactif, pour les essayer et les mettre au point. Le fichier `.zix` n'est pas produit si l'implémentation n'a pas d'interface explicite, puisque dans ce cas le module n'a pas de valeurs locales.

`-i` Force le compilateur à imprimer les types, les exceptions et les variables globales déclarés lors de la compilation d'une implémentation (fichier `.ml`). Ce peut être utile pour examiner les types trouvés par le compilateur. De plus, comme la sortie du compilateur obéit exactement à la syntaxe des interfaces de module, cette option aide à l'écriture de l'interface explicite (fichier `.mli`) d'un module : il suffit de rediriger la sortie standard du compilateur vers un fichier `.mli`, puis d'éditer ce fichier pour supprimer toutes les déclarations des globaux du module qu'on ne désire pas exporter.

`-I` *répertoire*
: Ajoute le répertoire donné à la liste des répertoires parcourus pour trouver les fichiers d'interface compilée des modules (`.zi`) et les fichiers de code compilé (`.zo`). Par défaut, le répertoire courant est parcouru le premier, puis le répertoire de la bibliothèque standard. Les répertoires ajoutés avec `-I` sont parcourus après le répertoire courant, mais avant le répertoire de la bibliothèque standard. Quand plusieurs répertoires sont ajoutés avec plusieurs options `-I` sur la ligne de commande, ces répertoires sont parcourus de la droite vers la gauche (le répertoire le plus à droite est parcouru le premier, le plus à gauche est parcouru le dernier). Les répertoires peuvent aussi être ajoutés au chemin d'accès directement dans le source des programmes avec la directive `#directory` (chapitre 2).

`-o` *nom-du-fichier-exécutable*
: Spécifie le nom du fichier produit par l'éditeur de liens.

Unix : Le nom par défaut est `a.out`, pour rester dans la tradition.

PC : Le nom par défaut est `CAML_OUT.EXE`.

Mac : Le nom par défaut est `Caml.Out`.

`-O` *ensemble-de-modules*
: Spécifie quel ensemble de modules standard doit être implicitement « ouvert » au début d'une compilation. Il y a trois ensembles de modules disponibles :

`cautious` (prudent)
: fournit les opérations courantes sur les entiers, flottants, caractères, chaînes, tableaux, ..., ainsi que la gestion d'exceptions, des entrées-sorties de base, etc. Les opérations de cet ensemble `cautious` font des tests de bornes lors des accès aux chaînes et aux tableaux, ainsi que de nombreuses vérifications de cohérence sur leurs arguments.

`fast` (rapide)
: fournit les mêmes opérations que l'ensemble `cautious`, mais sans vérifications. Les programmes compilés avec `-O fast` sont donc un peu plus rapides, mais ne sont pas sûrs.

`none` (rien)
: supprime toutes les ouvertures automatiques de modules. La compilation commence dans un environnement presque vide. Cette option n'est pas d'usage général, sauf pour compiler la bibliothèque standard elle-même.

Le mode de compilation par défaut est `-O cautious`. Voir le chapitre 9 pour une liste complète des modules des ensembles `cautious` et `fast`.

`-v` Imprime les numéros de version des différentes passes du compilateur.

3.3 Les modules et le système de fichier

Cette section a pour but de clarifier la relation entre les noms des modules et les noms des fichiers qui contiennent leur interface compilée et leur implémentation compilée.

Le compilateur déduit toujours le nom du module en cours de compilation en prenant le « nom de base » du fichier source (`.ml` ou `.mli`). C'est-à-dire qu'il supprime le nom du répertoire, s'il existe, ainsi que le suffixe `.ml` ou `.mli`. Les fichiers produits `.zi` et `.zo` ont le même nom de base que le fichier source ; donc, les fichiers compilés produits par le compilateur ont toujours leur nom de base identique au nom du module qu'ils décrivent (fichiers `.zi`) ou implémentent (fichiers `.zo`).

Pour les fichiers d'interface compilée (fichiers `.zi`), cet invariant doit impérativement être maintenu, puisque le compilateur se repose sur lui pour retrouver les fichiers d'interface des modules utilisés par le module en cours de compilation. Il est donc très risqué et généralement incorrect de renommer les fichiers `.zi`. Il est admissible de les déplacer dans un autre répertoire, si leur nom de base est préservé et si les options `-I` adéquates sont données au compilateur.

Les fichiers de code compilé (fichiers `.zo`), en revanche, peuvent être librement renommés une fois créés. D'abord parce que les fichiers `.zo` contiennent le vrai nom du module qu'ils définissent, si bien qu'il n'est pas nécessaire de déduire ce nom à partir du nom du fichier ; ensuite parce que l'éditeur de liens n'essaie jamais de trouver par lui-même le fichier `.zo` qui implémente un module de nom donné : il se repose sur l'utilisateur qui fournit lui-même la liste des fichiers `.zo`.

3.4 Erreurs courantes

Cette section décrit et explique les messages d'erreur les plus fréquents.

Cannot find file *nom-de-fichier*

(Impossible de trouver le fichier *nom-de-fichier*.)

Le fichier indiqué n'a pas été trouvé dans le répertoire courant, ni dans les répertoires du chemin d'accès. Le *nom-de-fichier* correspond soit à un fichier d'interface compilée (fichier `.zi`), soit à un fichier de code (fichier `.zo`). Si *nom-de-fichier* est de la forme *mod*`.zi`, cela signifie que vous tentez de compiler un fichier qui fait référence à des identificateurs du module *mod*, mais sans avoir préalablement compilé une interface pour le module *mod*. Remède : compilez *mod*`.mli` ou *mod*`.ml` d'abord, pour créer l'interface compilée *mod*`.zi`.

Si *nom-de-fichier* est de la forme *mod*`.zo`, cela veut dire que vous essayez de lier un fichier de code qui n'existe pas encore. Remède : compilez *mod*`.ml` d'abord.

Si votre programme est réparti sur plusieurs répertoires, cette erreur apparaît si vous n'avez pas spécifié ces répertoires dans le chemin d'accès. Remède : ajoutez les options `-I` adéquates sur la ligne de commande.

Corrupted compiled interface file *nom-de-fichier*

(Le fichier d'interface compilée *nom-de-fichier* est corrompu.)

Le compilateur produit cette erreur quand il lit un fichier d'interface compilée (fichier `.zi`) qui n'a pas la bonne structure. Cela signifie que l'écriture de ce fichier `.zi` n'a pas eu lieu correctement : par exemple, le disque était plein ou le compilateur a été interrompu au milieu de la création du fichier. Cette erreur intervient également quand un fichier `.zi` est modifié après sa création par le compilateur. Remède : effacez le fichier `.zi` corrompu et recréez-le.

Expression of type t_1 cannot be used with type t_2

(Cette expression, de type t_1, ne peut être utilisée avec le type t_2.)

C'est l'erreur de typage la plus fréquente. Le type t_1 est le type inféré pour l'expression (la partie du programme qui accompagne le message d'erreur), en examinant l'expression elle-même. Le type t_2 est le type attendu par le contexte de l'expression ; il est déduit par examen des utilisations de la valeur de cette expression dans le reste du programme. Si les deux types t_1 et t_2 sont incompatibles, le compilateur le signale par ce message d'erreur.

Dans certains cas, il est difficile de comprendre pourquoi les deux types t_1 et t_2 sont incompatibles. Par exemple, il arrive que le compilateur signale une expression de type `foo` qui ne peut pas être utilisée avec le type `foo`, et pourtant il semble bien que les deux types `foo` soient compatibles, puisqu'ils s'écrivent de la même façon. Ce n'est cependant pas toujours vrai. Deux constructeurs de types peuvent avoir le même nom, mais en fait représenter des types complètement différents. Cela arrive quand ce constructeur de type est redéfini. Exemple :

```
type foo = | A | B;;
let f = function | A -> 0 | B -> 1;;
type foo = | C | D;;
f C;;
```

Ce programme produit l'erreur `expression C of type foo cannot be used with type foo`.

Des types incompatibles avec le même nom peuvent également apparaître quand un module est modifié et recompilé, mais que certains de ses clients ne sont pas recompilés. Ce phénomène est dû au fait que les constructeurs de types qui apparaissent dans les fichiers `.zi` ne sont pas représentés par leurs noms (ce qui ne suffirait pas à les identifier à cause des redéfinitions de types), mais par des marques uniques qui leur sont attribuées quand la déclaration de type est compilée. Prenons pour exemple les trois modules :

```
mod1.ml:    type t = A | B;;
            let f = function A -> 0 | B -> 1;;

mod2.ml:    let g x = 1 + mod1__f(x);;

mod3.ml:    mod2__g mod1__A;;
```

Maintenant, supposons que `mod1.ml` est modifié et recompilé, mais que `mod2.ml` n'est pas recompilé. La recompilation de `mod1.ml` peut changer la marque attribuée au type `t`. Mais l'interface `mod2.zi` utilisera encore l'ancienne marque pour `mod1__t` dans le type de `mod2__g`. Donc, en compilant `mod3.ml`, le système

va se plaindre que le type de l'argument de `mod2__g` (c'est-à-dire `mod1__t` avec l'ancienne marque) n'est pas compatible avec le type de `mod1__A` (c'est-à-dire `mod1__t` avec la nouvelle marque). Remède : utilisez `make` ou un outil similaire pour être certain que tous les clients d'un module *mod* sont recompilés quand l'interface *mod*`.zi` change. Pour vérifier que le `Makefile` contient les bonnes dépendances, supprimez tous les fichiers `.zi` et reconstruisez tout le programme ; si aucune erreur `Cannot find file` ne se produit, tout va bien.

mod__nom **is referenced before being defined**

(*mod__nom* est mentionné avant d'avoir été défini.)

Cette erreur apparaît quand vous essayez de lier un ensemble de fichiers qui est incomplet ou mal ordonné. Vous avez par exemple oublié de fournir une implémentation pour le module nommé *mod* sur la ligne de commande (typiquement, le fichier nommé *mod*`.zo`, ou une bibliothèque contenant ce fichier). Remède : ajoutez le fichier `.ml` ou `.zo` qui manque sur la ligne de commande. Ou alors, vous avez fourni une implémentation pour le module nommé *mod*, mais elle arrive trop tard sur la ligne de commande : l'implémentation de *mod* doit être placée avant tous les fichiers qui font référence à une des variables globales qu'il définit. Remède : changez l'ordre des fichiers `.ml` et `.zo` sur la ligne de commande.

Bien sûr, vous aurez toujours cette erreur si vous écrivez des fonctions mutuellement récursives réparties sur plusieurs modules. Par exemple, si la fonction `mod1__f` appelle la fonction `mod2__g`, et la fonction `mod2__g` appelle la fonction `mod1__f`. Dans ce cas, quelle que soit la permutation des fichiers que vous puissiez faire sur la ligne de commande, le programme sera rejeté par l'éditeur de liens. Remèdes :

- Mettez `f` et `g` dans le même module.
- Paramétrez une fonction par l'autre. Ainsi, au lieu d'écrire :

```
mod1.ml:    let f x = ... mod2__g ... ;;
mod2.ml:    let g y = ... mod1__f ... ;;
```

  écrivez :

```
mod1.ml:    let f g x = ... g ... ;;
mod2.ml:    let rec g y = ... mod1__f g ... ;;
```

  et liez `mod1` avant `mod2`.

- Utilisez une référence qui contient l'une des deux fonctions, par exemple :

```
mod1.ml:    let ref_g =
                ref((fun x -> failwith "ref_g") : <type>);;
            let f x = ... !ref_g ... ;;
mod2.ml:    let g y = ... mod1__f ... ;;
            mod1__ref_g := g;;
```

Unavailable C primitive *f*

(La primitive C *f* n'est pas disponible.)

Cette erreur intervient quand vous essayez de lier du code qui appelle des fonctions externes écrites en C, alors que vous utilisez le mode « exécutant standard ». Comme expliqué dans le chapitre 8, vous devez utiliser dans ce cas le mode « exécutant dédié ». Remède : ajoutez l'option `-custom`, ainsi que les bibliothèques et fichiers objets qui implémentent vos fonctions externes.

4

Le système interactif (camllight)

Ce chapitre décrit le système interactif de Caml Light, qui permet une utilisation en direct au terminal du système Caml Light, grâce à une boucle sans fin de lecture-évaluation-impression. Dans ce mode, le système lit de manière répétée une phrase Caml Light, la type, la compile et l'évalue, puis imprime le type inféré et la valeur résultat, s'il y en a une. Le système imprime un `#` (dièse) en signe d'invite (en anglais *prompt*) avant de lire chaque phrase. Une phrase peut se prolonger sur plusieurs lignes et se termine obligatoirement par `;;` (le double point-virgule final).

Du point de vue du système de modules, toutes les phrases entrées interactivement sont traitées comme si elles constituaient l'implémentation d'un module unique nommé `top`. Donc, toutes les définitions interactives sont enregistrées dans le module `top`.

Unix : Le système interactif se lance par la commande `camllight`. Les phrases sont lues sur l'entrée standard, les résultats sont imprimés sur la sortie standard, les erreurs sur la sortie d'erreur standard. Un caractère fin-de-fichier sur l'entrée standard met fin au processus `camllight` (voir aussi la fonction `quit` à la section 4.2).

Le système interactif ne fait pas lui-même l'édition de ligne, mais il peut facilement être utilisé en conjonction avec un éditeur ligne externe tel que `fep` ; il suffit de lancer `fep -emacs camllight` ou `fep -vi camllight`.

À tout moment, l'analyse syntaxique, la compilation ou l'évaluation de la phrase courante peuvent être interrompues en appuyant sur `ctrl-C` (plus précisément en envoyant le signal `intr` au processus `camllight`). Une interruption vous ramène au signe d'invite (`#`).

Mac : Le système interactif est une application Macintosh indépendante nommée `Caml Light`. Cette application ne nécessite pas l'environnement MPW (Macintosh Programmer's Workshop).

Une fois lancée depuis le Bureau (en anglais *Finder*), l'application ouvre deux fenêtres, une fenêtre d'entrée intitulée « Caml Light Input » et une fenêtre de sortie intitulée « Caml Light Output ». Les phrases sont tapées dans la fenêtre « Caml Light Input ». La fenêtre « Caml Light Output » imprime une copie des phrases entrées au fur et à mesure de leur traitement par Caml Light, entrelacées avec les réponses du système. La touche « retour chariot » envoie le contenu de la fenêtre d'entrée à la boucle interactive. La touche « entrée » du

pavé numérique insère un saut de ligne sans envoyer le contenu de la fenêtre d'entrée. (On peut modifier ce comportement avec le menu « Preferences ».)

Le contenu de la fenêtre d'entrée peut être édité à tout moment, avec l'interface Macintosh habituelle. Le système conserve en mémoire les phrases précédemment entrées. On se déplace dans cet historique avec les menus « Previous entry » (touche commande-P) pour obtenir l'entrée précédente, et « Next entry » (touche commande-N) pour obtenir la suivante.

Pour quitter l'application `Caml Light`, vous pouvez ou bien sélectionner l'option « Quit » du menu « File », ou utiliser la fonction `quit` décrite à la section 4.2.

À tout moment, l'analyse syntaxique, la compilation ou l'évaluation de la phrase courante peuvent être interrompus en appuyant sur « commande-point », ou en sélectionnant l'option « Interrupt Caml Light » dans le menu « Caml Light ». Une interruption vous ramène au signe d'invite (`#`).

PC : Le système interactif se lance par la commande `caml`. (Le nom complet `camllight` a dû être tronqué à cause de limitations bien connues de MS-DOS.) Les phrases sont lues sur l'entrée standard, les résultats sont imprimés sur la sortie standard, les erreurs sur la sortie d'erreur standard. La fonction système `quit` met fin à la commande `caml`.

Plusieurs éditeurs lignes résidents, qui ajoutent l'édition de ligne et des facilités d'historique à l'interpréteur de commandes `command.com`, fonctionnent aussi avec `caml`. Quelques éditeurs qui peuvent collaborer avec `caml` : `dosedit`, `ced`, `toddy`. Mais `doskey` de MS-DOS 5.0 est inutilisable.

Avec la version 8086, la lecture, la compilation ou l'évaluation de la phrase courante peuvent être interrompues en appuyant sur `ctrl-break`. Ceci vous ramène au signe d'invite (`#`). Appuyer sur `ctrl-C` interrompt l'entrée, mais n'arrête pas une phrase qui ne fait pas d'entrées-sorties.

Avec la version 80386, appuyer sur `ctrl-C` ou `ctrl-break` interrompt le système à tout moment : pendant la lecture aussi bien que pendant la compilation ou l'évaluation.

4.1 Options

Les options suivantes peuvent apparaître sur la ligne de commande de `camllight` ou `caml`.

`-g` Lance le système interactif en mode de mise au point (en anglais *debugging*). Ce mode donne accès aux valeurs et aux types locaux à un module, c'est-à-dire non exportés dans l'interface du module. En mode normal, ces objets locaux sont inaccessibles (toute tentative pour y accéder provoque une erreur « Variable non liée ») ; en mode de mise au point, ces objets locaux deviennent visibles, au même titre que les objets exportés dans l'interface du module. En particulier, les valeurs de types abstraits sont affichées avec leur représentation et les fonctions locales à un module peuvent être « tracées » (voir la fonction `trace`, section 4.2). Ceci ne fonctionne que pour les modules compilés en mode de mise au point (soit avec l'option `-g` du compilateur indépendant, soit avec la fonction `compile` du système interactif en mode de mise au point), c'est-à-dire les modules pour lesquels existe un fichier `.zix`.

`-I` *répertoire*
: Ajoute le répertoire donné à la liste des répertoires parcourus pour trouver les fichiers d'interface des modules compilés (`.zi`) et les fichiers de code compilé (`.zo`). Par défaut, le répertoire courant est parcouru le premier, puis le répertoire de la bibliothèque standard. Les répertoires ajoutés avec `-I` sont parcourus après le répertoire courant, mais avant le répertoire de la bibliothèque standard. Quand plusieurs répertoires sont ajoutés avec plusieurs options `-I` sur la ligne de commande, ces répertoires sont parcourus de la droite vers la gauche (le répertoire le plus à droite est parcouru le premier, le plus à gauche est parcouru le dernier). Les répertoires peuvent aussi être ajoutés au chemin d'accès après que l'on a lancé le système interactif avec la directive `#directory` (voir le chapitre 2).

`-O` *ensemble-de-modules*
: Spécifie quel ensemble de modules standard doit être implicitement « ouvert » au lancement du système interactif. Il y a trois ensembles de modules disponibles :

  `cautious` (prudent)
  : fournit les opérations habituelles sur les entiers, flottants, caractères, chaînes, tableaux, ... ainsi que la gestion d'exceptions, des entrées-sorties de base, etc. Les opérations de cet ensemble `cautious` font des tests de bornes lors des accès aux chaînes et aux tableaux, ainsi que de nombreuses vérifications de cohérence sur leurs arguments.

  `fast` (rapide)
  : fournit les mêmes opérations que l'ensemble `cautious`, mais sans tests. Les phrases compilées avec `-O fast` sont donc un peu plus rapides, mais ne sont plus sûres.

`none` (rien)
: supprime toutes les ouvertures automatiques de modules. La compilation commence dans un environnement presque vide. Cette option n'est pas d'usage général.

Par défaut, le système interactif se lance dans le mode `-O cautious`. Voir le chapitre 9 pour une liste complète des modules des ensembles `cautious` et `fast`.

4.2 Fonctions de contrôle du système interactif

Le module `toplevel` de la bibliothèque standard, ouvert par défaut quand le système interactif est lancé, fournit un certain nombre de fonctions qui contrôlent le comportement du système interactif, chargent des fichiers en mémoire et tracent l'exécution des programmes.

`quit ()`
: Quitte la boucle interactive et met fin à la commande Caml Light.

`include "`*nom-de-fichier*`"`
: Lit, compile et exécute les phrases source du fichier *nom-de-fichier*. L'extension `.ml` est automatiquement ajoutée à *nom-de-fichier*, si elle en était absente. L'inclusion est textuelle : les phrases sont traitées comme si elles avaient été tapées sur l'entrée standard. En particulier, les identificateurs globaux définis par ces phrases sont enregistrés dans le module nommé `top`, et non pas dans un nouveau module.

`load "`*nom-de-fichier*`"`
: Charge en mémoire le code source de l'implémentation d'un module. Lit, compile et exécute les phrases source du fichier *nom-de-fichier*. L'extension `.ml` est automatiquement ajoutée à *nom-de-fichier*, si elle en était absente. La fonction `load` se comporte presque comme `include`, sauf qu'un nouveau module est créé, avec pour nom le nom de base de *nom-de-fichier*. Les identificateurs globaux définis dans le fichier *nom-de-fichier* sont enregistrés dans ce module, au lieu du module `top` dans le cas de la fonction `include`. Par exemple, en supposant que le fichier `foo.ml` contient la seule phrase

```
let bar = 1;;
```

exécuter `load "foo"` définit l'identificateur `foo__bar` avec pour valeur 1.

Attention : le module chargé n'est pas automatiquement ouvert : l'identificateur `bar` n'est donc pas complété automatiquement en `foo__bar`. Pour obtenir ce résultat, vous devez encore exécuter la directive `#open "foo"` après avoir chargé le fichier `foo.ml`.

`compile "`*nom-de-fichier*`"`
: Compile une implémentation ou une interface de module (fichier `.ml` ou `.mli`). La compilation se déroule exactement comme avec le compilateur indépendant : tout se passe comme si on avait fait `camlc -c` *nom-de-fichier*. Si le système interactif est en mode de mise au point (option `-g` du système interactif, ou

fonction `debug_mode` ci-dessous), la compilation s'effectue elle aussi en mode de mise au point, comme avec l'option `-g` du compilateur indépendant. Le résultat de la compilation est laissé dans des fichiers (`.zo`, `.zi`, `.zix`). Le code compilé n'est pas chargé en mémoire.

`load_object "`*nom-de-fichier*`"`
: Charge en mémoire le code compilé contenu dans le fichier *nom-de-fichier*. L'extension `.zo` est automatiquement ajoutée à *nom-de-fichier*, si elle en était absente. Le fichier de code a été produit soit par le compilateur indépendant `camlc`, soit par la commande `compile` du système interactif. Les identificateurs globaux définis dans le fichier *nom-de-fichier* sont donc enregistrés dans leur propre module, pas dans le module `top`, comme pour la fonction `load`.

`trace "`*nom-de-fonction*`"`
: Après l'exécution de `trace "foo"`, tous les appels à la fonction globale nommée `foo` vont être « tracés ». Cela signifie que l'argument et le résultat sont imprimés pour chaque appel, ainsi que les exceptions qui s'échappent de `foo`, déclenchées soit par `foo` elle-même, soit par une des fonctions appelées depuis `foo`. Si `foo` est une fonction curryfiée, chaque argument est imprimé quand il est passé à la fonction.

`untrace "`*nom-de-fonction*`"`
: Exécuter `untrace "foo"` supprime le mécanisme de trace pour la fonction globale nommée `foo`.

`debug_mode` *drapeau-booléen*
: Passe le système interactif en mode de mise au point (`debug_mode true`), ou revient en mode standard (`debug_mode false`). Exécuter `debug_mode true` équivaut à lancer le système interactif avec l'option `-g`. En mode de mise au point, les objets locaux (non exportés dans l'interface) d'un module deviennent visibles au même titre que les objets exportés. Voir la description de l'option `-g` à la section 4.1.

`verbose_mode` *drapeau-booléen*
: Après `verbose_mode true`, les compilations faites avec `compile` affichent les objets définis par le module, avec les types inférés, comme le fait l'option `-i` du compilateur indépendant. Les compilations redeviennent silencieuses après `verbose_mode false`.

`cd "`*nom-de-répertoire*`"`
: Fixe le répertoire de travail courant.

`gc ()`
: Termine le cycle de récupération mémoire (en anglais *garbage collection*) en cours et renvoie la quantité d'espace libre dans le tas mémoire, en octets. Si vous désirez accomplir un cycle de récupération mémoire complet, appelez cette fonction deux fois de suite.

4.3 Le système interactif et le système de modules

Les phrases tapées interactivement peuvent faire référence à des identificateurs définis dans d'autres modules que le module `top`, en utilisant les mêmes mécanismes que pour les modules compilés séparément : en utilisant soit des identificateurs qualifiés (`nomDeModule__nomLocal`), soit des identificateurs non qualifiés qui sont automatiquement complétés en examinant la liste des modules ouverts (comme décrit dans la section 1.2 du chapitre 1). Les modules ouverts au lancement sont listés dans la documentation de la bibliothèque standard (chapitre 9). D'autres modules peuvent être ouverts par la suite avec la directive `#open`.

Dans tous les cas, avant de faire référence à une variable globale d'un module différent du module `top`, une définition de cette variable globale doit préalablement avoir été mise en mémoire. Au départ, le système interactif contient les définitions de tous les identificateurs de la bibliothèque standard. Les définitions des modules de l'utilisateur peuvent être introduites avec les fonctions `load` ou `load_object` décrites à la section 4.2. Faire référence à une variable globale pour laquelle aucune définition n'a été fournie par `load` ou `load_object` provoque l'erreur « `Identifier ... is referenced before being defined` » (« l'identificateur ... est mentionné avant d'avoir été défini »). Pour bien faire comprendre ces deux points, qui rendent l'utilisation interactive des modules un peu délicate, voyons quelques exemples typiques.

1. La fonction de bibliothèque `sub_string` est définie dans le module `string`. Ce module fait partie de la bibliothèque standard et c'est un des modules automatiquement ouverts au lancement. Donc, les deux phrases

   ```
   sub_string "qwerty" 1 3;;
   string__sub_string "qwerty" 1 3;;
   ```

   sont correctes, sans nécessité d'utiliser `#open`, `load` ou `load_object`.

2. La fonction de bibliothèque `printf` est définie dans le module `printf`. Ce module fait partie de la bibliothèque standard, mais il n'est pas automatiquement ouvert au lancement. Donc, la phrase

   ```
   printf__printf "%s %s" "hello" "world";;
   ```

   est correctement exécutée, tandis que

   ```
   printf "%s %s" "hello" "world";;
   ```

   produit l'erreur « `Variable printf is unbound` » (« la variable `printf` n'est pas liée ») puisque aucun des modules ouverts pour l'instant ne définit un global avec pour nom local `printf`. Cependant,

   ```
   #open "printf";;
   printf "%s %s" "hello" "world";;
   ```

   marche sans problème.

3. Supposez que le fichier `foo.ml` se trouve dans le répertoire courant et contienne la seule phrase

   ```
   let x = 1;;
   ```

   Quand le système interactif démarre, toute référence à `x` produira l'erreur « `Variable x is unbound` » (« la variable `x` n'est pas liée »). Une référence à `foo__x` produira l'erreur « `Cannot find file foo.zi` » (« impossible de trouver le fichier `foo.zi` »), puisque le vérificateur de types cherche à charger l'interface compilée du module `foo` pour trouver le type de `x`. Pour charger en mémoire le module `foo`, faites simplement :

   ```
   load "foo";;
   ```

   Les références à `foo__x` seront alors typées et évaluées correctement. Puisque `load` n'ouvre pas le module qu'il charge, les références à `x` échoueront encore avec l'erreur « `Variable x is unbound` » (« la variable `x` n'est pas liée »). Vous devrez explicitement taper

   ```
   #open "foo";;
   ```

   pour que l'identificateur `x` soit automatiquement complété en `foo__x`.

4. Finalement, supposons que le fichier `foo.ml` précédent ait été préalablement compilé avec la commande `camlc -c` ou la fonction prédéfinie `compile`. Le répertoire courant contient donc une interface compilée `foo.zi`, indiquant que `foo__x` est une variable globale de type `int`, ainsi qu'un fichier de code `foo.zo`, qui définit `foo__x` avec pour valeur `1`. Quand le système interactif démarre, toute référence à `foo__x` produira l'erreur « `Identifier foo__x is referenced before being defined` » (« l'identificateur `foo__x` est référencé avant d'avoir été défini »). Contrairement au cas 3, le vérificateur de types a réussi à trouver l'interface compilée du module `foo`. Mais l'exécution ne peut pas avoir lieu, puisque aucune définition pour `foo__x` n'a été mise en mémoire. Pour ce faire, exécutez :

   ```
   load_object "foo";;
   ```

   Cette commande charge en mémoire le fichier `foo.zo`, et donc définit `foo__x`. Les références à `foo__x` sont alors correctement évaluées. Comme dans le cas 3, les références à `x` échouent encore, car `load_object` n'ouvre pas le module qu'il charge. Encore une fois, vous devrez explicitement taper

   ```
   #open "foo";;
   ```

   pour que l'identificateur `x` soit automatiquement complété en `foo__x`.

4.4 Erreurs courantes

Cette section décrit et explique les messages d'erreur les plus fréquents.

Cannot find file *nom-de-fichier*
(Impossible de trouver le fichier *nom-de-fichier*.)

Le fichier indiqué n'a pas été trouvé dans le répertoire courant, ni dans les répertoires du chemin d'accès.

Si *nom-de-fichier* est de la forme *mod*`.zi`, cela veut dire que la phrase courante fait référence à des identificateurs du module *mod*, mais que vous n'avez pas encore compilé d'interface pour ce module *mod*. Remèdes : ou bien chargez le fichier *mod*`.ml` (avec la fonction `load`), ce qui construira en mémoire l'interface compilée du module *mod* ; ou bien utilisez `camlc` pour compiler *mod*`.mli` ou *mod*`.ml`, de façon à créer l'interface compilée *mod*`.zi` avant de lancer le système interactif.

Si *nom-de-fichier* est de la forme *mod*`.zo`, cela veut dire que vous tentez de charger avec `load_object` un fichier de code qui n'existe pas encore. Remèdes : compilez *mod*`.ml` avec `camlc` avant de lancer le système interactif, ou bien utilisez `load` au lieu de `load_object` pour charger le code source au lieu d'un fichier compilé.

Si *nom-de-fichier* est de la forme *mod*`.ml`, cela veut dire que `load` ou `include` ne peut pas trouver le fichier source spécifié. Remède : vérifiez l'orthographe du nom du fichier, ou écrivez-le s'il n'existe pas.

mod`__`*nom* **is referenced before being defined**

(*mod*`__`*nom* est référencé avant d'avoir été défini.)

Vous avez oublié de charger en mémoire une implémentation de module, avec `load` ou `load_object`. Ce point a été expliqué de manière détaillée dans la section 4.3.

Corrupted compiled interface file *nom-de-fichier*

(Le fichier d'interface compilée *nom-de-fichier* est corrompu.)

Voir la section 3.4.

Expression of type t_1 cannot be used with type type t_2

(Cette expression, de type t_1, ne peut être utilisée avec le type t_2.)

Voir la section 3.4.

4.5 Construction de systèmes interactifs dédiés : camlmktop

La commande `camlmktop` construit des systèmes Caml Light interactifs qui contiennent du code de l'utilisateur préchargé au lancement.

Mac : Cette commande n'est pas disponible dans la version Macintosh.

PC : Cette commande n'est pas disponible dans les versions PC.

La commande `camlmktop` prend pour argument un ensemble de fichiers `.zo` et les lie avec les fichiers de code qui implémentent le système interactif de Caml Light. L'utilisation typique est :

```
camlmktop -o mytoplevel foo.zo bar.zo gee.zo
```

ce qui crée le fichier de code `mytoplevel`, contenant le système interactif de Caml Light plus le code des trois fichiers `.zo`. Pour lancer ce système interactif, donnez-le en argument à la commande `camllight` :

```
camllight mytoplevel
```

Ceci commence une session interactive normale, si ce n'est que le code de `foo.zo`, `bar.zo` et `gee.zo` est déjà chargé en mémoire, comme si vous aviez tapé :

```
load_object "foo";;
load_object "bar";;
load_object "gee";;
```

à l'entrée de la session. Les modules `foo`, `bar` et `gee` ne sont cependant pas ouverts ; vous devez encore taper vous-même

```
#open "foo";;
```

si c'est ce que vous désirez.

4.6 Options

Les options suivantes peuvent apparaître sur la ligne de commande de `camlmktop`.

`-custom`
: Édition de liens en mode « exécutant dédié ». Voir l'option correspondante de `camlc`, dans le chapitre 3.

`-I` *répertoire*
: Ajoute le répertoire donné à la liste des répertoires parcourus pour trouver les fichiers de code compilé (`.zo`).

`-o` *nom-du-fichier-exécutable*
: Spécifie le nom du fichier produit par l'éditeur de liens. Le nom par défaut est `camltop.out`.

5
L'exécutant (camlrun)

La commande `camlrun` exécute les fichiers de code produits par la phase d'édition de lien de la commande `camlc`.

Mac : Cette commande est un outil MPW, et non pas une application Macintosh indépendante.

5.1 Vue d'ensemble

La commande `camlrun` comprend trois parties principales : l'interpréteur de code de la machine virtuelle de Caml Light, qui exécute effectivement les fichiers de code ; l'allocateur de mémoire et le récupérateur de mémoire (en anglais *garbage collector*) ; et enfin un ensemble de fonctions C qui implémentent les opérations primitives telles que les entrées-sorties.

L'appel de `camlrun` se fait comme suit :

`camlrun` *options code-exécutable* arg_1 arg_2 … arg_n

Le premier argument non optionnel est considéré comme le nom du fichier qui contient le code exécutable. (Ce fichier est cherché dans le chemin d'accès des exécutables et dans le répertoire courant.) Les arguments suivants sont passés au programme Caml Light, dans le tableau de chaînes `sys__command_line`. L'élément 0 de ce tableau est le nom du fichier de code exécutable ; les éléments 1 à n sont les autres arguments de la ligne de commande : arg_1 à arg_n.

Comme mentionné dans le chapitre 3, dans la plupart des cas, les fichiers de code exécutables produits par la commande `camlc` sont auto-exécutables et savent lancer la commande `camlrun` eux-mêmes. C'est-à-dire que si `caml.out` est un fichier de code exécutable,

`caml.out` arg_1 arg_2 … arg_n

se comporte comme

`camlrun caml.out` arg_1 arg_2 … arg_n

Notez qu'il est impossible de passer des options à `camlrun` en invoquant directement `caml.out`.

5.2 Options

Les options suivantes peuvent apparaître sur la ligne de commande de `camlrun`. (Il y a des options supplémentaires pour contrôler le comportement du récupérateur de mémoire, mais elles ne sont pas destinées à l'utilisateur normal.)

`-v` Force `camlrun` à imprimer des messages de diagnostics relatifs à l'allocation et à la récupération mémoire. Utile pour corriger les problèmes liés à la mémoire.

5.3 Erreurs courantes

Cette section décrit et explique les messages d'erreur les plus fréquents.

nom-de-fichier`: no such file or directory`
(*nom-de-fichier*: aucun fichier ou répertoire de ce nom.)

Si *nom-de-fichier* est le nom d'un fichier de code auto-exécutable, cela signifie soit que le fichier n'existe pas, soit qu'il ne parvient pas à lancer l'interpréteur de code `camlrun` sur lui-même. Dans le deuxième cas, Caml Light n'a pas été correctement installé sur votre machine.

`Cannot exec camlrun` (Impossible de lancer `camlrun`.)
(Au lancement d'un fichier de code auto-exécutable.) La commande `camlrun` n'a pas été trouvée dans le chemin d'accès des exécutables. Vérifiez que Caml Light a été installé correctement sur votre machine.

`Cannot find the bytecode file` (Fichier de code introuvable.)
Le fichier que `camlrun` tente d'exécuter (par exemple le fichier donné en premier argument non optionnel à `camlrun`) ou bien n'existe pas, ou bien n'est pas un fichier de code exécutable valide.

`Truncated bytecode file` (Fichier de code tronqué.)
Le fichier que `camlrun` tente d'exécuter n'est pas un fichier de code exécutable valide. Il a probablement été tronqué ou modifié depuis sa création. Supprimez-le et refaites-le.

`Uncaught exception` (Exception non rattrapée.)
Le programme exécuté déclenche une exception inattendue : il déclenche une exception à un moment donné et cette exception n'est jamais rattrapée par une construction `try`...`with`. C'est l'arrêt immédiat du programme. Si vous souhaitez connaître l'exception qui s'échappe, utilisez la fonction `printexc__f` de la bibliothèque standard (et n'oubliez pas de lier votre programme avec l'option `-g`).

`Out of memory` (La mémoire est pleine.)
Le programme exécuté requiert plus de mémoire que la mémoire disponible. Ou bien le programme construit des structures de données trop grosses, ou bien il effectue trop d'appels de fonctions imbriqués, et la pile déborde. Dans certains cas, votre programme est parfaitement correct, il demande seulement plus de mémoire que votre machine n'en peut fournir. (Ce qui arrive fréquemment sur les petits micro-ordinateurs, mais rarement sur les machines Unix.) Dans d'autres cas, le message « out of memory » révèle une erreur dans votre programme : une fonction

récursive qui ne termine pas, une allocation de chaîne ou de tableau excessivement gros, ou encore une tentative de construire une structure de données infinie.

Pour vous aider à diagnostiquer ce genre d'erreur, lancez votre programme avec l'option `-v` de `camlrun`. Si vous voyez apparaître de nombreux messages « `Growing stack` ... » (« Extension de la pile ... »), c'est probablement une fonction récursive qui boucle. Si les messages qui apparaissent sont de nombreux « `Growing heap` ... » (« Extension du tas ... ») et que la taille du tas augmente lentement, c'est probablement que vous essayez de construire une structure de données avec trop de cellules (une infinité ?). Si au contraire vous voyez apparaître peu de messages « `Growing heap` ... », mais que la taille du tas augmente beaucoup à chaque fois, c'est probablement que vous demandez la construction d'une chaîne ou d'un tableau trop grands.

6
Le gestionnaire de bibliothèques (camllibr)

Mac : Cette commande est un outil MPW, et non pas une application Macintosh indépendante.

6.1 Vue d'ensemble

Le programme `camllibr` réunit dans un seul fichier un ensemble de fichiers de code compilé (fichiers `.zo`). Le fichier résultat est aussi un fichier de code compilé et possède aussi l'extension `.zo`. On peut le passer à la phase d'édition de liens du compilateur `camlc` en lieu et place de l'ensemble des fichiers de code d'origine. Autrement dit, après l'exécution de

```
camllibr -o bibliothèque.zo mod1.zo mod2.zo mod3.zi mod4.zo
```

tous les appels à l'éditeur de liens de la forme

```
camlc ... bibliothèque.zo ...
```

sont exactement équivalents à

```
camlc ... mod1.zo mod2.zo mod3.zi mod4.zo ...
```

La principale utilisation de `camllibr` est de construire une bibliothèque composée de plusieurs modules : de cette façon, les utilisateurs de la bibliothèque n'ont qu'un fichier `.zo` à spécifier sur la ligne de commande de `camlc`, au lieu de toute une série de fichiers `.zo`, un par module contenu dans la bibliothèque.

La phase d'édition de liens de `camlc` est assez intelligente pour supprimer le code correspondant à des phrases inutilisées : en particulier, les définitions de variables globales qui ne servent pas après leur définition. Donc, il n'y a aucun problème à mettre plusieurs modules, même des modules rarement utilisés, en une seule bibliothèque : cela ne va pas augmenter la taille des exécutables.

L'appel à `camllibr` se fait ainsi :

`camllibr` *options fichier*$_1$`.zo` ... *fichier*$_n$`.zo`

où *fichier₁*.zo à *fichierₙ*.zo sont les fichiers objets à réunir. L'ordre dans lequel ces noms de fichiers apparaissent sur la ligne de commande est significatif : les phrases compilées contenues dans la bibliothèque vont être exécutées dans cet ordre. (On rappelle que c'est une erreur de faire référence à une variable globale qui n'a pas encore été définie.)

6.2 Options

Les options suivantes peuvent apparaître sur la ligne de commande de `camllibr`.

`-files` *fichier-de-réponses*
: Traite les fichiers dont les noms sont listés dans le fichier *fichier-de-réponses*, comme si ces noms apparaissaient sur la ligne de commande. Les noms de fichiers contenus dans *fichier-de-réponses* sont séparés par des blancs (espaces, tabulations ou sauts de ligne). Cette option permet de circonvenir de sottes limitations sur la longueur de la ligne de commande.

`-I` *répertoire*
: Ajoute le répertoire donné à la liste des répertoires parcourus pour trouver les fichiers arguments (`.zo`). Par défaut, le répertoire courant est parcouru le premier, puis le répertoire de la bibliothèque standard. Les répertoires ajoutés avec `-I` sont parcourus après le répertoire courant, mais avant le répertoire de la bibliothèque standard. Quand plusieurs répertoires sont ajoutés avec plusieurs options `-I` sur la ligne de commande, ces répertoires sont parcourus de la droite vers la gauche (le répertoire le plus à droite est parcouru le premier, le plus à gauche est parcouru le dernier).

`-o` *nom-de-bibliothèque*
: Spécifie le nom du fichier produit. Le nom par défaut est `library.zo`.

6.3 Transformer du code en une bibliothèque

Pour écrire une bibliothèque, il est généralement plus facile de la couper en plusieurs modules qui reflètent la structure interne de la bibliothèque. Du point de vue des utilisateurs de la bibliothèque, en revanche, il est préférable de voir la bibliothèque comme un seul module, avec seulement un fichier d'interface (fichier `.zi`) et un fichier d'implémentation (fichier `.zo`) : l'édition de liens est simplifiée, et il est inutile de mettre toute une série de directives `#open`, comme de se souvenir de la structure interne de la bibliothèque.

La commande `camllibr` permet donc de n'avoir qu'un seul fichier `.zo` pour toute la bibliothèque. Voici comment utiliser le système de module de Caml Light pour avoir un seul fichier `.zi` pour toute la bibliothèque. Pour être plus concrets, supposons que la bibliothèque comprenne trois modules : `windows`, `images` et `buttons`. L'idée est d'ajouter un quatrième module, `mylib`, qui réexporte les parties publiques de `windows`, `images` et `buttons`. L'interface `mylib.mli` contient des définitions pour les types qui sont publics (exportés avec leurs définitions), des déclarations pour les types qui sont abstraits (exportés sans leurs définitions), et des déclarations pour les fonctions qui peuvent être appelées depuis le code de l'utilisateur :

```
(* File mylib.mli *)
type 'a option = None | Some of 'a;;    (* un type public *)
type window and image and button;;      (* trois types abstraits *)
value new_window : int -> int -> window (* les fonctions publiques *)
  and draw_image : image -> window -> int -> int -> unit
  and ...
```

L'implémentation du module `mylib` fait simplement correspondre les types abstraits et les fonctions publiques aux types et fonctions correspondants des modules `windows`, `images` et `buttons` :

```
(* File mylib.ml *)
type window == windows__win
 and image  == images__pixmap
 and button == buttons__t;;
let new_window = windows__open_window
and draw_image = images__draw
and ...
```

Les fichiers `windows.ml`, `images.ml` et `buttons.ml` peuvent ouvrir le module `mylib`, pour avoir accès aux types publics définis dans l'interface `mylib.mli`, tels que le type `option`. Bien sûr, ces modules ne doivent pas faire référence aux types abstraits ni aux fonctions publiques, pour éviter les circularités.

Les types tels que `windows__win` dans l'exemple précédent peuvent être exportés par le module `windows`, soit abstraitement soit concrètement (avec leurs définitions). Souvent, il est nécessaire de les exporter concrètement, parce que les autres modules de la bibliothèque (`images`, `buttons`) ont besoin de construire ou de déstructurer directement des valeurs de ces types. Même si `windows__win` est exporté concrètement par le module `windows`, ce type restera abstrait à l'utilisateur de la bibliothèque, puisqu'il est abstrait par l'interface publique de `mylib`.

La construction de la bibliothèque `mylib` se déroule comme suit :

```
camlc -c mylib.mli       # crée mylib.zi
camlc -c windows.mli windows.ml images.mli images.ml
camlc -c buttons.mli buttons.ml
camlc -c mylib.ml        # crée mylib.zo
mv mylib.zo tmplib.zo    # renommage pour éviter de détruire mylib.zo
camllibr -o mylib.zo windows.zo images.zo buttons.zo tmplib.zo
```

Puis on copie `mylib.zi` et `mylib.zo` en un endroit accessible aux utilisateurs de la bibliothèque. Les autres fichiers `.zi` et `.zo` n'ont pas besoin d'être copiés.

7
Générateurs d'analyseurs syntaxiques et lexicaux (camllex, camlyacc)

Ce chapitre décrit deux générateurs de programmes : `camllex`, qui produit un analyseur lexical à partir d'un ensemble d'expressions rationnelles (en anglais *regular expressions*) accompagnées d'actions sémantiques, et `camlyacc`, qui produit un analyseur syntaxique à partir d'une grammaire accompagnée d'actions sémantiques.

Ces générateurs de programmes sont très proches de `lex` et `yacc`, commandes Unix bien connues, qui existent dans la plupart des environnements de programmation C. Ce chapitre suppose une connaissance certaine de `lex` et `yacc` : bien qu'il décrive la syntaxe d'entrée de `camllex` et `camlyacc` et les principales différences avec `lex` et `yacc`, il n'explique pas comment écrire une spécification d'analyseur pour `lex` ou `yacc`. Les lecteurs qui ne sont pas familiers avec `lex` et `yacc` sont invités à se reporter à *Compilateurs : principes, techniques et outils*, de Aho, Sethi et Ullman (InterÉditions), *Compiler design in C*, de Holub (Prentice Hall), ou *Lex & Yacc*, de Levine, Mason et Brown (O'Reilly).

Les flux et le filtrage de flux offrent un autre moyen d'écrire des analyseurs syntaxiques et lexicaux. La technique du filtrage de flux est plus puissante que la combinaison de `camllex` et `camlyacc` dans certains cas (analyseurs syntaxiques d'ordre supérieur), mais moins puissante dans d'autres (priorités). Choisissez l'approche la mieux adaptée à votre problème d'analyse particulier.

Mac : Ces commandes sont des outils MPW, et non pas des applications Macintosh indépendantes.

86 : Ces commandes ne sont pas disponibles dans la version PC 8086.

7.1 Vue d'ensemble de camllex

La commande `camllex` produit un analyseur lexical à partir d'un ensemble d'expressions rationnelles (en anglais *regular expressions*) annotées par des actions sémantiques, dans le style de `lex`. En supposant que le fichier d'entrée est *lexer*`.mll`,

l'exécution de

`camllex` *lexer*`.mll`

produit du code Caml Light pour un analyseur lexical dans le fichier *lexer*`.ml`. Ce fichier définit une fonction d'analyse lexicale par point d'entrée de la définition de l'analyseur lexical. Ces fonctions ont les même noms que les points d'entrée. Les fonctions d'analyse lexicale prennent en argument un tampon d'analyse lexicale et renvoient l'attribut sémantique du point d'entrée correspondant. Les tampons d'analyse lexicale appartiennent à un type de données abstrait implémenté dans le module `lexing` de la bibliothèque standard. Les fonctions `create_lexer_channel`, `create_lexer_string` et `create_lexer` du module `lexing` créent des tampons d'analyse lexicale qui lisent à partir d'un canal d'entrée, d'une chaîne de caractères ou d'une fonction de lecture quelconque, respectivement. (Voir la description du module `lexing`, section 10.5.)

Quand elles sont utilisées avec un analyseur syntaxique engendré par `camlyacc`, les actions sémantiques calculent une valeur qui appartient au type `token` défini par le module d'analyse syntaxique engendré. (Voir la description de `camlyacc` à la section 7.3.)

7.2 Syntaxe des définitions d'analyseurs lexicaux

Les définitions d'analyseur lexicaux se présentent comme suit :

```
{ prélude }
rule point d'entrée =
  parse expression-rationnelle { action }
      | ...
      | expression-rationnelle { action }
and point d'entrée =
  parse ...
and ...
;;
```

Les commentaires sont délimités par `(*` et `*)`, comme en Caml Light.

7.2.1 En-tête

La section « prélude » est formée d'un texte Caml Light arbitraire entouré d'accolades. Il peut être omis. S'il est présent, le texte entouré est copié tel quel au début du fichier produit. Typiquement, cette section d'en-tête contient les directives `#open` nécessaires aux actions, et éventuellement quelques fonctions auxiliaires utilisées dans les actions.

7.2.2 Points d'entrée

Le noms des points d'entrée doivent être des identificateurs Caml Light valides.

7.2.3 Expressions rationnelles

Les expressions rationnelles sont analogues à celles de `lex`, avec une syntaxe un peu plus proche de celle de Caml.

` *char* `
: Un caractère littéral, avec la même syntaxe qu'en Caml Light. Reconnaît le caractère correspondant au littéral.

_
: Reconnaît n'importe quel caractère.

`eof`
: Reconnaît une condition de fin de fichier sur l'entrée de l'analyseur lexical.

" *chaîne* "
: Une chaîne littérale, avec la même syntaxe qu'en Caml Light. Reconnaît la suite de caractères correspondante.

[*ensemble-de-caractères*]
: Reconnaît tout caractère appartenant à l'ensemble de caractères donné. Les ensembles de caractères valides sont : un caractère littéral ` *c* `, représentant l'ensemble à un seul élément *c* ; un intervalle ` $c_1$ ` - ` $c_2$ `, représentant tous les caractères entre c_1 et c_2 inclus ; la concaténation de plusieurs ensembles, représentant l'union de ces ensembles.

[^ *ensemble-de-caractères*]
: Reconnaît n'importe quel caractère unique n'appartenant pas à l'ensemble de caractères donné.

expr *
: (Répétition.) Reconnaît les concaténations de zéro, une ou plusieurs chaînes qui sont reconnues par *expr*.

expr +
: (Répétition stricte.) Reconnaît les concaténations d'une ou plusieurs chaînes qui sont reconnues par *expr*.

expr ?
: (Option.) Reconnaît soit la chaîne vide, soit une chaîne reconnue par *expr*.

$expr_1$ | $expr_2$
: (Alternative.) Reconnaît les chaînes reconnues par $expr_1$ ou par $expr_2$.

$expr_1$ $expr_2$
: (Concaténation.) Reconnaît les concaténations de deux chaînes, la première reconnue par $expr_1$, la seconde par $expr_2$.

(*expr*)
: Reconnaît les mêmes chaînes que *expr*.

Les opérateurs `*` et `+` ont la plus forte priorité ; viennent ensuite `?`, puis la concaténation et enfin `|` (l'alternative).

7.2.4 Actions

Les actions sont des expressions Caml Light arbitraires. Elles sont évaluées dans un contexte où l'identificateur `lexbuf` est lié au tampon d'analyseur lexical courant. Voici quelques utilisations typiques de `lexbuf`, en conjonction avec les opérations sur tampons fournies par le module `lexing` de la bibliothèque standard.

`lexing__get_lexeme lexbuf`
: Renvoie la chaîne reconnue.

`lexing__get_lexeme_char lexbuf` *n*
: Renvoie le $n^{\text{ième}}$ caractère de la chaîne reconnue. Le premier caractère correspond à $n = 0$.

`lexing__get_lexeme_start lexbuf`
: Renvoie la position absolue dans le texte d'entrée du début de la chaîne reconnue. Le premier caractère lu dans le texte d'entrée a la position 0.

`lexing__get_lexeme_end lexbuf`
: Renvoie la position absolue dans le texte d'entrée de la fin de la chaîne reconnue. Le premier caractère lu dans le texte d'entrée a la position 0.

point d'entrée `lexbuf`
: (Où *point d'entrée* est le nom d'un autre point d'entrée dans la même définition d'analyseur lexical.) Appelle récursivement l'analyseur lexical sur le point d'entrée donné. Utile pour faire l'analyse lexicale de commentaires emboîtés, par exemple.

7.3 Vue d'ensemble de camlyacc

La commande `camlyacc` produit un analyseur syntaxique à partir d'une spécification de grammaire algébrique (en anglais *context-free grammar*) annotée par des actions sémantiques, dans le style de `yacc`. En supposant que le fichier d'entrée est *grammaire*`.mly`, l'exécution de

`camlyacc` *options grammaire*`.mly`

produit du code Caml Light pour un analyseur syntaxique dans le fichier *grammaire*`.ml`, et son interface dans le fichier *grammaire*`.mli`.

Le module engendré définit une fonction d'analyse syntaxique par point d'entrée de la grammaire. Ces fonctions ont les mêmes noms que les points d'entrée. Les fonctions d'analyse syntaxique prennent pour arguments un analyseur lexical (une fonction des tampons d'analyseurs syntaxiques vers les lexèmes) et un tampon d'analyseur lexical, et renvoient l'attribut sémantique du point d'entrée correspondant. Les fonctions d'analyse lexicale sont généralement produites à partir d'une spécification d'analyseur lexical par le programme `camllex`. Les tampons d'analyse lexicale appartiennent à un type de données abstrait implémenté dans le module `lexing` de la bibliothèque standard. Les lexèmes sont des valeurs du type somme `token`, défini dans l'interface du fichier *grammaire*`.mli` produit par `camlyacc`.

7.4 Syntaxe des définitions de grammaires

Les définitions de grammaires ont le format suivant :

`%{`
  prélude
`%}`
  déclarations
`%%`
  règles
`%%`
  épilogue

Les commentaires sont entourés par `/*` et `*/`, comme en C.

7.4.1 Prélude et épilogue

Les sections prélude et épilogue sont du code Caml Light qui est copié tel quel dans le fichier *grammaire*`.ml`. Ces deux sections sont optionnelles. Le prélude est copié au début du fichier produit, l'épilogue à la fin. Typiquement, le prélude contient les directives `#open` nécessaires aux actions, et éventuellement quelques fonctions auxiliaires utilisées dans les actions ; l'épilogue peut contenir un point d'entrée dans le programme, comme par exemple une boucle d'interaction. Tous les globaux définis dans le prélude et l'épilogue restent locaux au module *grammaire*, puisqu'il n'y a pas moyen de les déclarer dans l'interface *grammaire*`.mli`. En particulier, si les points d'entrée calculent des valeurs de certains types concrets, ces types ne peuvent pas être déclarés dans le prélude, mais doivent être déclarés dans un autre module, que l'on ouvre par `#open` dans le prélude de la grammaire.

7.4.2 Déclarations

Chaque déclaration occupe une ligne. Elles commencent toutes par un caractère `%`.

`%token` *symbole . . . symbole*
: Déclare les symboles donnés comme lexèmes (symboles terminaux). Ces symboles sont ajoutés comme constructeurs constants au type somme `token`.

`%token <` *type* `>` *symbole . . . symbole*
: Déclare les symboles donnés comme lexèmes munis d'un attribut du type donné. Ces symboles sont ajoutés au type somme `token` comme des constructeurs avec un argument du type donné. La partie *type* est une expression de type Caml Light arbitraire, à ceci près que tous les constructeurs de type non prédéfinis doivent avoir des noms complètement qualifiés (de la forme `nomDeModule__nomDeType`), même si les bonnes directives `#open` directives (en l'occurrence `#open "nomDeModule"`) apparaissent dans la section prélude. En effet, le prélude est copié seulement dans le fichier `.ml`, mais pas dans le fichier `.mli`, alors que la partie *type* d'une déclaration `%token` est copiée dans le `.mli`.

`%start` *symbole . . . symbole*
: Déclare les symboles donnés comme points d'entrée de la grammaire. Pour

chaque point d'entrée, une fonction d'analyse syntaxique du même nom est définie dans le module produit. Les non-terminaux qui ne sont pas déclarés comme points d'entrée n'ont pas de fonction d'analyse syntaxique associée. Il faut impérativement donner un type aux points d'entrée avec la directive `%type` suivante.

`%type` < *type* > *symbole* ... *symbole*

Spécifie le type de l'attribut sémantique des symboles donnés. Ce n'est obligatoire que pour les points d'entrée. Il n'est pas nécessaire de préciser le type des autres non-terminaux : ces types seront inférés par le compilateur Caml Light à la compilation du fichier produit (sauf si l'option `-s` est active). La partie *type* est une expression de type Caml Light arbitraire, à ceci près que tous les constructeurs de type non prédéfinis doivent avoir des noms complètement qualifiés (de la forme `nomDeModule__nomDeType`), même si les bonnes directives `#open` (en l'occurrence `#open "nomDeModule"`) apparaissent dans la section prélude. En effet, le prélude est copié seulement dans le fichier `.ml`, mais pas dans le fichier `.mli`, alors que la partie *type* d'une déclaration `%type` est copiée dans le `.mli`.

`%left` *symbole* ... *symbole*
`%right` *symbole* ... *symbole*
`%nonassoc` *symbole* ... *symbole*

Associe une priorité et une associativité aux symboles donnés. Tous les symboles d'une même ligne reçoivent la même priorité. Cette priorité est plus forte que celle des symboles précédemment déclarés par un `%left`, un `%right` ou un `%nonassoc`. En revanche elle est plus faible que celle des symboles déclarés ensuite par un `%left`, un `%right` ou un `%nonassoc`.

Les symboles sont déclarés associatifs à gauche (`%left`) ou à droite (`%right`), ou non associatifs (`%nonassoc`). Ces symboles sont la plupart du temps des lexèmes ; ils peuvent aussi être des symboles ne correspondant à aucun non-terminal de la grammaire, mais utilisés dans des directives `%prec`.

7.4.3 Règles

La syntaxe des règles est sans surprises :

non-terminal :
    symbole ... *symbole* { *action-sémantique* }
  | ...
  | *symbole* ... *symbole* { *action-sémantique* }
;

La directive `%prec` *symbole* peut apparaître dans le membre droit d'une règle, pour outrepasser la priorité et l'associativité par défaut de la règle en leur substituant celles du symbole donné.

Les actions sémantiques sont des expressions Caml Light arbitraires, qui sont évaluées pour produire l'attribut sémantique attaché au non-terminal défini. Les actions sémantiques peuvent accéder aux attributs sémantiques des symboles du membre droit de leur règle avec la notation `$`. `$1` est l'attribut du premier symbole (le plus à gauche), `$2` est l'attribut du second symbole, etc.

Les actions intervenant au milieu des règles ne sont pas permises. La récupération d'erreur n'est pas implémentée.

7.5 Options

Les options suivantes sont reconnues par `camlyacc`.

`-v` Produit une description des tables de l'analyseur syntaxique et un compte rendu des conflits engendrés par les ambiguïtés de la grammaire. Cette description est écrite dans le fichier *grammaire*`.output`.

`-s` Produit un fichier *grammaire*`.ml` découpé en phrases plus petites. Les actions sémantiques sont recopiées dans le fichier *grammaire*`.ml` sous la forme d'un tableau de fonctions. Par défaut, ce tableau est construit en une seule phrase. Pour des grammaires de grande taille, ou contenant des actions sémantiques complexes, la phrase en question peut requérir une grande quantité de mémoire pendant sa compilation par Caml Light. Avec l'option `-s`, le tableau des actions est construit incrémentalement, en compilant une phrase par action. Ceci permet d'abaisser les besoins en mémoire du compilateur, mais il n'est plus possible d'inférer le type des symboles non terminaux : la vérification de type est supprimée pour les symboles dont le type n'est pas spécifié par une directive `%type`.

`-b` *préfixe*
Nomme les fichiers de sortie *préfixe*`.ml`, *préfixe*`.mli`, *préfixe*`.output`, au lieu des noms par défaut.

7.6 Un exemple complet

En guise d'exemple, voici comment programmer une calculatrice « quatre opérations ». Ce programme lit des expressions arithmétiques sur l'entrée standard, une par ligne, et imprime leurs valeurs. Voici la définition de la grammaire :

```
/* Fichier de l'analyseur syntaxique: parser.mly */
%token <int> INT
%token PLUS MOINS FOIS DIV
%token PARENG PAREND
%token FINDELIGNE
%right PLUS MOINS       /* priorité la plus faible */
%right FOIS DIV         /* priorité intermédiaire */
%nonassoc MOINSUNAIRE   /* priorité la plus forte */
%start ligne            /* le point d'entrée */
%type <int> ligne
%%
ligne:
    expr FINDELIGNE              { $1 }
;
expr:
    INT                          { $1 }
  | PARENG expr PAREND           { $2 }
  | expr PLUS expr               { $1 + $3 }
  | expr MOINS expr              { $1 - $3 }
  | expr FOIS expr               { $1 * $3 }
  | expr DIV expr                { $1 / $3 }
  | MOINS expr %prec MOINSUNAIRE { - $2 }
```

```
        ;
```

Voici la définition de l'analyseur lexical correspondant :

```
(* Fichier lexer.mll *)
{
#open "parser";;   (* Le type token est défini dans *)
                   (* l'analyseur syntaxique parser.mli *)
exception Fin_de_fichier;;
}
rule lexeme = parse
    [` ` `\t`]     { lexeme lexbuf } (* supprime les blancs *)
  | [`\n` ]        { FINDELIGNE }
  | [`0`-`9`]+     { INT(int_of_string (get_lexeme lexbuf)) }
  | `+`            { PLUS }
  | `-`            { MOINS }
  | `*`            { FOIS }
  | `/`            { DIV }
  | `(`            { PARENG }
  | `)`            { PAREND }
  | eof            { raise Fin_de_fichier }
;;
```

Et voici le programme principal, qui combine l'analyseur syntaxique et l'analyseur lexical :

```
(* Fichier calc.ml *)
try
  let tampon = lexing__create_lexer_channel std_in in
  while true do
    let résultat = parser__ligne lexer__lexeme tampon in
      print_int résultat; print_newline(); flush std_out
  done
with lexer__Fin_de_fichier -> ();;
```

Pour compiler le tout, exécutez :

```
camllex lexer.mll       # engendre lexer.ml
camlyacc parser.mly     # engendre parser.ml et parser.mli
camlc -c parser.mli
camlc -c lexer.ml
camlc -c parser.ml
camlc -c calc.ml
camlc -o calc lexer.zo parser.zo calc.zo
```

8
Communication entre Caml Light et C

Ce chapitre décrit comment des primitives, écrites en C par l'utilisateur, peuvent être ajoutées à l'exécutant de Caml Light et appelées à partir du code Caml Light. Il s'adresse aux utilisateurs très avertis et peut être sauté en première lecture.

Mac : Cette facilité n'est pas implémentée dans la version Macintosh.

PC : Cette facilité n'est pas implémentée dans les versions PC.

8.1 Vue d'ensemble

8.1.1 Déclaration des primitives

Les primitives sont déclarées dans un module interface (un fichier `.mli`), de la même façon qu'une valeur Caml Light normale, sauf que la déclaration est suivie d'un signe `=`, de l'arité de la fonction (nombre d'arguments) et du nom de la fonction C correspondante. Par exemple, voici comment la primitive `input` est déclarée dans l'interface du module `io` de la bibliothèque standard :

```
value input : in_channel -> string -> int -> int -> int
            = 4 "input"
```

Les primitives ayant plusieurs arguments sont toujours curryfiées. La fonction C n'a pas obligatoirement le même nom que la fonction Caml Light.

Les valeurs ainsi déclarées primitives dans une interface de module ne doivent pas être implémentées dans l'implémentation du module (le fichier `.ml`). Elles sont directement utilisables dans l'implémentation du module.

8.1.2 Implémentation des primitives

Les primitives ayant au plus cinq arguments sont implémentées par des fonctions C qui prennent des arguments de type `value` et renvoient un résultat de type `value`. Le type `value` est le type des représentations de valeurs Caml Light. Il encode les types de base (entiers, flottants, chaînes, ...), ainsi que les structures de données Caml Light. Le type `value` et les fonctions de conversion et macros associées sont décrits en détail par

la suite. Par exemple, voici la déclaration de la fonction C qui implémente la primitive `input` :

```
value input(channel, buffer, offset, length)
        value channel, buffer, offset, length;
{
 ...
}
```

Quand la fonction primitive est appliquée dans un programme Caml Light, la fonction C est appelée avec les valeurs des expressions auxquelles la primitive est appliquée. La valeur retournée par la fonction est repassée au programme Caml Light en résultat de l'application de la fonction.

Les primitives de l'utilisateur ayant six arguments ou plus sont implémentées par des fonctions C qui reçoivent deux arguments : un pointeur vers un tableau de valeurs Caml Light (les valeurs des arguments) et un entier qui est le nombre d'arguments fournis à la primitive :

```
value prim_avec_7_arguments(argv, argn)
        value * argv;
        int argn;
{
  ... argv[0] ...;              /* Le premier argument */
  ... argv[6] ...;              /* Le septième argument */
}
```

L'implémentation d'une primitive utilisateur comprend en fait deux parties distinctes : un travail d'encodage-décodage d'une part, puisqu'à l'appel il faut décoder les arguments Caml Light pour en extraire des valeurs C tandis qu'au retour il faut encoder la valeur C résultat en une valeur Caml Light, et d'autre part le calcul du résultat escompté à partir des arguments traduits en valeurs C. Sauf pour les primitives très simples, il vaut mieux écrire deux fonctions C distinctes pour réaliser ces deux tâches distinctes. L'une des fonctions implémente à proprement parler la primitive, travaillant complètement en C, donc ayant des valeurs C en arguments et résultat. La seconde fonction, qu'on appelle enveloppe (en anglais *stub code*), est un simple enrobage de la première : elle se contente de convertir les arguments Caml Light qu'elle a reçus en valeurs C pour appeler la première fonction, puis convertit le résultat C retourné en une valeur Caml Light. Par exemple, voici l'enveloppe pour la primitive `input` :

```
value input(channel, buffer, offset, length)
        value channel, buffer, offset, length;
{
  return Val_long(getblock((struct channel *) channel,
                           &Byte(buffer, Long_val(offset)),
                           Long_val(length)));
}
```

(Ici, `Val_long`, `Long_val`, etc., sont des macros de conversion sur le type `value`, qu'on expliquera plus tard.) Le travail effectif est accompli par la fonction `getblock`, qui est déclarée ainsi :

```
long getblock(channel, p, n)
     struct channel * channel;
```

```
          char * p;
          long n;
{
  ...
}
```

Pour écrire du code C qui opère sur des valeurs Caml Light, on dispose des fichiers d'en-tête (en anglais *headers*) suivants :

`mlvalues.h`	définition du type `value` et des macros de conversion
`alloc.h`	fonctions d'allocation (pour créer des objets Caml Light structurés)
`memory.h`	diverses fonctions relatives à la mémoire (pour les modifications en place de structures, etc.).

Ces fichiers résident dans le répertoire de la bibliothèque standard de Caml Light (`/usr/local/lib/caml-light` dans l'installation standard).

8.1.3 Lier du code C avec du code Caml Light

L'exécutant de Caml Light comprend trois parties principales : l'interpréteur de code, le gestionnaire mémoire et un ensemble de fonctions C qui implémentent les opérations primitives. Des instructions de la machine abstraite permettent d'appeler ces fonctions C, désignées par leur indice dans une table de fonctions (la table des primitives).

Dans son mode par défaut, l'éditeur de liens de Caml Light produit du code pour l'exécutant standard, avec un ensemble standard de primitives. Les références à des primitives qui ne sont pas dans cet ensemble standard conduisent à l'erreur « `unavailable C primitive` » (primitive C non disponible).

Dans le mode « exécutant dédié », l'éditeur de liens de Caml Light parcourt les fichiers de code (fichiers `.zo`) et détermine l'ensemble des primitives nécessaires. Puis il construit un exécutant adéquat, par un appel à l'éditeur de liens de la machine, avec :

- la table des primitives nécessaires,
- une bibliothèque qui fournit l'interpréteur de code, le gestionnaire mémoire et les primitives standard,
- les bibliothèques et fichiers de code objet (fichiers `.o` et `.a`) mentionnés sur la ligne de commande de l'éditeur de liens de Caml Light, qui fournissent les implémentations des primitives de l'utilisateur.

On construit ainsi un exécutant contenant toutes les primitives nécessaires. L'éditeur de liens de Caml Light produit alors du code pour cet exécutant dédié. Ce code est copié à la fin de l'exécutant dédié, de telle manière qu'il soit automatiquement exécuté quand le fichier produit (exécutant dédié plus code) est lancé.

Pour faire l'édition de liens en mode « exécutant dédié », exécutez la commande `camlc` avec :

- l'option `-custom`,
- les noms des fichiers objets Caml Light (fichiers `.zo`) souhaités,

- les noms des fichiers objets et bibliothèques C (fichiers `.o` et `.a`) qui implémentent les primitives utilisées. (On peut aussi spécifier les bibliothèques avec la syntaxe `-l` habituelle, par exemple `-lX11` au lieu de `/usr/lib/libX11.a`.)

8.2 Le type `value`

Tous les objets Caml Light sont représentés par le type C `value`, défini dans le fichier `mlvalues.h`, avec les macros qui manipulent les valeurs de ce type. Un objet de type `value` est soit :

- un entier non alloué (en anglais *unboxed*),
- un pointeur vers un bloc dans le tas (en anglais *heap*) (de tels blocs sont alloués grâce à l'une des fonctions `alloc_*` de la section 8.4.4),
- un pointeur vers un objet en dehors du tas (par exemple, un pointeur vers un bloc alloué par `malloc`, ou vers une variable C).

8.2.1 Valeurs entières

Les valeurs entières encodent des entiers signés 31 bits. Elles ne sont pas allouées.

8.2.2 Blocs

Les blocs dans le tas qui sont devenus inutiles sont récupérés par le récupérateur de mémoire (en anglais *garbage collector*) ; ils obéissent donc à des contraintes de structure très strictes. Chaque bloc comprend un en-tête contenant la taille du bloc (en mots) et un marqueur. Le marqueur indique comment est structuré le contenu du bloc. Un marqueur plus petit que `No_scan_tag` désigne un bloc structuré, contenant des valeurs bien formées, qui est récursivement examiné par le récupérateur de mémoire. Un marqueur supérieur ou égal à `No_scan_tag` indique un bloc brut, dont le contenu n'est pas examiné par le récupérateur de mémoire. Pour autoriser l'implémentation de primitives polymorphes « ad hoc » telles que l'égalité ou les entrées-sorties de valeurs structurées, les blocs sont en outre classés selon leur marqueur :

Marqueur	Contenu du bloc
0 à `No_scan_tag` − 1	Un bloc structuré (un tableau d'objets Caml Light). Chaque champ est un objet de type `value`.
`Closure_tag`	Une fermeture (en anglais *closure*) qui représente une valeur fonctionnelle. Le premier mot est un pointeur vers un morceau de code, le second est une `value` contenant l'environnement.
`String_tag`	Une chaîne de caractères.
`Double_tag`	Un flottant en double précision.
`Abstract_tag`	Un bloc d'un type de données abstrait.
`Final_tag`	Un bloc pour un type de données abstrait avec une fonction de « finalisation », qui est appelée quand le bloc est libéré.

8.2.3 Pointeurs en dehors du tas

Tout pointeur vers l'extérieur du tas peut sans problèmes être converti par un *cast* vers le type `value`, et réciproquement. Cela inclut les pointeurs retournés par `malloc` et les pointeurs vers des variables C obtenus par l'opérateur `&`.

8.3 Représentation des types de données Caml Light

Cette section décrit le codage des types de données Caml Light dans le type `value`.

8.3.1 Types atomiques

`int`	Valeurs entières non allouées.
`char`	Valeurs entières non allouées (code ASCII).
`float`	Blocs avec marqueur `Double_tag`.
`string`	Blocs avec marqueur `String_tag`.

8.3.2 Types produits

Les n-uplets et les tableaux sont représentés par des pointeurs vers des blocs, avec marqueur 0.

Les enregistrements sont aussi représentés par des blocs marqués zéro. L'ordre de définition des étiquettes de l'enregistrement lors de la définition du type détermine l'indice des champs de l'enregistrement : la valeur associée à l'étiquette déclarée en premier est enregistrée dans le champ 0 du bloc, la valeur associée à l'étiquette déclarée en second va dans le champ 1, et ainsi de suite.

8.3.3 Types somme

Les valeurs d'un type somme sont représentés par des blocs dont le marqueur code le constructeur. Les constructeurs d'un type somme donné sont numérotés à partir de zéro jusqu'au nombre de constructeurs moins un, suivant l'ordre de leur apparition dans la définition du type somme. Les constructeurs constants sont représentés par blocs de taille zéro (atomes), marqués par le numéro du constructeur. Les constructeurs non constants dont l'argument est un n-uplet sont représentés par un bloc de taille n, marqué par le numéro du constructeur ; les n champs contiennent les composants du n-uplet argument. Les autres constructeurs non constants sont représentés par un bloc de taille 1, marqué par le numéro du constructeur ; le champ 0 contient la valeur de l'argument du constructeur. Exemple :

Terme construit	Représentation
`()`	Taille = 0, marqueur = 0
`false`	Taille = 0, marqueur = 0
`true`	Taille = 0, marqueur = 1
`[]`	Taille = 0, marqueur = 0
`h::t`	Taille = 2, marqueur = 1, premier champ = `h`, second champ = `t`

8.4 Opérations sur les valeurs

8.4.1 Tests de type

- `Is_int`(v) est vrai si la valeur v est un entier non alloué, faux sinon.
- `Is_block`(v) est vrai si la valeur v est un pointeur vers un bloc, et faux si c'est un entier non alloué.

8.4.2 Opérations sur les entiers

- `Val_long`(l) renvoie la valeur qui code le `long int` l.
- `Long_val`(v) renvoie le `long int` codé dans la valeur v.
- `Val_int`(i) renvoie la valeur qui code l'`int` i.
- `Int_val`(v) renvoie l'`int` codé dans la valeur v.

8.4.3 Accès aux blocs

- `Wosize_val`(v) renvoie la taille de la valeur v, en mots, en-tête non compris.
- `Tag_val`(v) renvoie le marqueur de la valeur v.
- `Field`(v, n) renvoie la valeur contenue dans le $n^{\text{ième}}$ champ du bloc structuré v. Les champs sont numérotés de 0 à `Wosize_val`(v) $- 1$.
- `Code_val`(v) renvoie la partie code de la fermeture v.
- `Env_val`(v) renvoie la partie environnement de la fermeture v.
- `string_length`(v) renvoie la longueur (nombre de caractères) de la chaîne v.
- `Byte`(v, n) renvoie le $n^{\text{ième}}$ caractère de la chaîne v, avec pour type `char`. Les caractères sont numérotés de 0 à `string_length`(v) $- 1$.
- `Byte_u`(v, n) renvoie le $n^{\text{ième}}$ caractère de la chaîne v, avec pour type `unsigned char`. Les caractères sont numérotés de 0 à `string_length`(v) $- 1$.
- `String_val`(v) renvoie un pointeur vers le premier caractère de la chaîne v, avec pour type `char *`. Ce pointeur est une chaîne C valide : il y a toujours un caractère nul après le dernier caractère de la chaîne. Malgré tout, les chaînes de Caml Light peuvent contenir des caractères nuls n'importe où, ce qui trouble généralement les fonctions C travaillant sur les chaînes de caractères.
- `Double_val`(v) renvoie le flottant contenu dans la valeur v, avec pour type `double`.

Les expressions `Field`(v, n), `Byte`(v, n), `Byte_u`(v, n) et `Double_val`(v) sont des valeurs assignables (en anglais *l-values*) valides. On peut donc les affecter, ce qui produit une modification physique de la valeur v. Les affectations directes à `Field`(v, n) doivent être faites avec prudence pour ne pas troubler le récupérateur de mémoire (voir la section 8.5).

8.4.4 Allocation de blocs

Du point de vue des fonctions d'allocation, les blocs sont groupés selon leur taille, en blocs de taille zéro, petits blocs (taille inférieure ou égale à `Max_young_wosize`), et gros blocs (taille supérieure à `Max_young_wosize`). À chaque classe de taille de bloc sa fonction d'allocation. La constante `Max_young_wosize` est déclarée dans le fichier `mlvalues.h`. Sa valeur est au moins 64 (mots), si bien que tout bloc de taille constante inférieure ou égale à 64 est considéré comme petit. Pour les blocs dont la taille est calculée à l'exécution, il faut comparer cette taille avec `Max_young_wosize` pour déterminer la bonne procédure d'allocation.

- `Atom`(t) renvoie un « atome » (bloc de taille zéro) avec marqueur t. Les blocs de taille zéro sont préalloués en dehors du tas. Il est incorrect d'essayer d'allouer un bloc de taille zéro avec les fonctions de la section 8.4.4. Par exemple, `Atom`(0) représente `()`, `false` et `[]` ; `Atom`(1) représente `true`. (Pour plus de clarté, `mlvalues.h` définit des macros `Val_unit`, `Val_false` et `Val_true`.)
- `alloc`(n, t) renvoie un nouveau petit bloc de taille $n \leq$ `Max_young_wosize` mots, avec marqueur t. Si ce bloc est un bloc structuré (autrement dit si $t <$ `No_scan_tag`), alors les champs du bloc (qui contiennent initialement des bits aléatoires) doivent être initialisés avec des valeurs légales (en utilisant l'affectation directe aux champs du bloc) avant la prochaine allocation.
- `alloc_tuple`(n) renvoie un nouveau petit bloc de taille $n \leq$ `Max_young_wosize` mots, avec marqueur 0. Les champs de ce bloc doivent être initialisés avec des valeurs légales avant la prochaine allocation ou modification.
- `alloc_shr`(n, t) renvoie un nouveau bloc de taille n, avec marqueur t. La taille du bloc peut être plus grande que `Max_young_wosize` (ou plus petite, mais dans ce cas il est plus efficace d'appeler `alloc` au lieu de `alloc_shr`). Si ce bloc est un bloc structuré (c'est-à-dire si $t <$ `No_scan_tag`), alors les champs du bloc (qui contiennent initialement des bits aléatoires) doivent être initialisés avec des valeurs légales (à l'aide de la fonction `initialize`) avant la prochaine allocation.
- `alloc_string`(n) renvoie une chaîne de caractères de longueur n. La chaîne contient initialement des bits aléatoires.
- `copy_string`(s) renvoie une valeur chaîne de caractères contenant une copie de s (une chaîne C terminée par un caractère nul).
- `copy_double`(d) renvoie une valeur flottante initialisée avec le `double` d.
- `alloc_array`(f, a) alloue un tableau de valeurs et appelle la fonction f sur chaque élément du tableau a pour le transformer en valeur. Le tableau a est un tableau de pointeurs terminé par le pointeur nul. La fonction f reçoit chaque pointeur pour argument et renvoie une valeur. Le bloc marqué zéro retourné par `alloc_array`(f, a) est rempli avec les valeurs retournées par les appels successifs à f.
- `copy_string_array`(p) alloue un tableau de chaînes, copiées à partir du pointeur vers un tableau de chaînes p (un `char **`).

8.4.5 Déclenchement d'exceptions

Les fonctions C ne peuvent pas déclencher n'importe quelle exception. Cependant on fournit deux fonctions qui déclenchent deux exceptions standard :

- `failwith`(s), où s est une chaîne C terminée par un caractère nul (de type `char *`), déclenche l'exception `Failure` avec pour argument s.
- `invalid_argument`(s), où s est une chaîne C terminée par un caractère nul (de type `char *`), déclenche l'exception `Invalid_argument` avec pour argument s.

8.5 Vivre en harmonie avec le récupérateur de mémoire

Les blocs inutilisés du tas sont automatiquement récupérés par le récupérateur de mémoire. Cela demande un peu de coopération de la part du code C qui manipule les blocs alloués dans le tas. Voici trois règles de base qu'il doit respecter.

Règle 1 *Après l'allocation d'un bloc structuré (un bloc avec marqueur inférieur à* `No_scan_tag`*), tous les champs de ce bloc doivent être remplis avec des valeurs bien formées avant la prochaine opération d'allocation. Si le bloc a été alloué avec* `alloc` *ou* `alloc_tuple`*, le remplissage a lieu par affectation directe aux champs du bloc :*

$$\texttt{Field}(v, n) = v_n;$$

Si le bloc a été alloué avec `alloc_shr`*, le remplissage s'opère par la fonction* `initialize` *:*

$$\texttt{initialize}(\&\texttt{Field}(v, n), v_n);$$

La prochaine allocation peut déclencher un cycle de récupération mémoire (en anglais *garbage collection*). Le récupérateur de mémoire suppose toujours que tous les blocs structurés contiennent des valeurs bien formées. Lors de leur allocation, les blocs nouvellement créés contiennent des données aléatoires, qui généralement ne constituent pas des valeurs bien formées.

Si vous avez vraiment besoin d'allouer avant que tous les champs puissent recevoir leur valeur finale, initialisez d'abord avec une valeur constante (par exemple `Val_long(0)`), puis allouez, et modifiez les champs avec les valeurs correctes (voir règle 3).

Règle 2 *Les variables locales contenant des valeurs doivent être enregistrées auprès du récupérateur de mémoire (en utilisant les macros* `Push_roots` *et* `Pop_roots`*), si elles doivent survivre à un appel à une fonction d'allocation.*

L'enregistrement se fait par les macros `Push_roots` et `Pop_roots`. L'appel `Push_roots`(r, n) déclare un tableau r de n valeurs et les enregistre auprès du récupérateur de mémoire. Les valeurs contenues dans $r[0]$ à $r[n-1]$ sont traitées comme des « racines » par le récupérateur de mémoire. Une valeur racine a les propriétés suivantes : si elle pointe vers un bloc alloué dans le tas, ce bloc et son contenu ne seront pas récupérés ; de plus, si ce bloc est déplacé par le récupérateur de

mémoire, la valeur racine est modifiée pour pointer vers la nouvelle adresse du bloc. `Push_roots`(r, n) doit apparaître dans un bloc de programme C exactement entre la dernière déclaration de variable locale et la première instruction du bloc. Pour annuler l'enregistrement des racines, il faut appeler la macro `Pop_roots`() avant de quitter le bloc C contenant `Push_roots`(r, n). (L'enregistrement des racines est automatiquement annulé si une exception Caml est déclenchée.)

Règle 3 *L'affectation directe d'un champ d'un bloc, comme dans*

`Field`$(v, n) = v'$;

n'est sûre que si v est un bloc nouvellement alloué par `alloc` *ou* `alloc_tuple`*, c'est-à-dire si aucune allocation n'a lieu entre l'allocation de v et l'affectation du champ. Dans tous les autres cas, ne faites jamais d'affectation directe. Si le bloc vient d'être alloué par* `alloc_shr`*, utilisez* `initialize` *pour affecter une valeur à un champ pour la première fois :*

`initialize`(&`Field`$(v, n), v'$);

Sinon, si vous voulez modifier un champ qui contenait déjà une valeur bien formée, appelez la fonction `modify` *:*

`modify`(&`Field`$(v, n), v'$);

Pour illustrer les règles précédentes, voici une fonction C qui construit et retourne une liste formée des deux entiers reçus en paramètres :

```
value alloc_list_int(i1, i2)
        int i1, i2;
{
  value result;
  Push_roots(r, 1);
  r[0] = alloc(2, 1);               /* Alloue une cellule de liste */
  Field(r[0], 0) = Val_int(i2);     /* car = l'entier i2 */
  Field(r[0], 1) = Atom(0);         /* cdr = la liste vide [] */
  result = alloc(2, 1);             /* Alloue l'autre cellule */
  Field(result, 0) = Val_int(i1);   /* car = l'entier i1 */
  Field(result, 1) = r[0];          /* cdr = la première cellule */
  Pop_roots();
  return result;
}
```

La cellule de liste allouée la première doit survivre à l'allocation de la deuxième ; la valeur retournée par le premier appel à `alloc` doit donc être mise dans une racine enregistrée. La valeur retournée par le second appel à `alloc` peut résider dans la variable locale non enregistrée `result`, puisque nous n'allouons plus dans le reste de la fonction.

Dans cet exemple, la liste est construite de bas en haut. On peut aussi la construire de haut en bas. C'est moins efficace, mais cela nous permet d'illustrer l'usage de la fonction `modify`.

```
value alloc_list_int(i1, i2)
        int i1, i2;
{
```

```
  value tail;
  Push_roots(r, 1);
  r[0] = alloc(2, 1);               /* Alloue une cellule de liste */
  Field(r[0], 0) = Val_int(i1);     /* car = l'entier i1 */
  Field(r[0], 1) = Val_int(0);      /* Une valeur quelconque */
  tail = alloc(2, 1);               /* Alloue l'autre cellule */
  Field(tail, 0) = Val_int(i2);     /* car = l'entier i2 */
  Field(tail, 1) = Atom(0);         /* cdr = la liste vide [] */
  modify(&Field(r[0], 1), tail);    /* cdr du résultat = tail */
  Pop_roots();
  return r[0];
}
```

Il serait incorrect d'écrire directement `Field(r[0], 1) = tail`, parce que l'allocation de `tail` a eu lieu après l'allocation de `r[0]`.

8.6 Un exemple complet

Cette section explique dans les grandes lignes l'interfaçage de programmes Caml Light avec les fonctions de la bibliothèque Unix `curses`. Commençons par l'interface de module `curses.mli` qui déclare les primitives et structures de données de la bibliothèque `curses` :

```
type window;;                 (* Le type "window" demeure abstrait *)
value initscr: unit -> window = 1 "curses_initscr"
  and endwin: unit -> unit = 1 "curses_endwin"
  and refresh: unit -> unit = 1 "curses_refresh"
  and wrefresh : window -> unit = 1 "curses_wrefresh"
  and newwin: int -> int -> int -> int -> window = 4 "curses_newwin"
  and mvwin: window -> int -> int -> unit = 3 "curses_mvwin"
  and addch: char -> unit = 1 "curses_addch"
  and mvwaddch: window -> int -> int -> char -> unit =
              4 "curses_mvwaddch"
  and addstr: string -> unit = 1 "curses_addstr"
  and mvwaddstr: window -> int -> int -> string -> unit =
              4 "curses_mvwaddstr"
;; (* beaucoup d'autres fonctions omises *)
```

On compile cette interface par :

```
        camlc -c curses.mli
```

Pour implémenter ces fonctions, il nous faut seulement écrire le code enveloppe : les fonctions de base sont déjà implémentées dans la bibliothèque `curses`. Le fichier de code enveloppe, `curses.c`, a l'allure suivante :

```
#include <curses.h>
#include <mlvalues.h>

value curses_initscr(unit)
        value unit;
{
  return (value) initscr();
  /* On a le droit de convertir directement de WINDOW * à value,
```

```
     puisque le résultat de initscr() est un bloc créé par malloc() */
}

value curses_wrefresh(win)
        value win;
{
  wrefresh((value) win);
  return Val_unit;    /* Ou Atom(0) */
}

value curses_newwin(nlines, ncols, x0, y0)
        value nlines, ncols, x0, y0;
{
  return (value) newwin(Int_val(nlines), Int_val(ncols),
                        Int_val(x0), Int_val(y0));
}

value curses_addch(c)
        value c;
{
  addch(Int_val(c));  /* Les caractères sont codés par des entiers */
  return Val_unit;
}

value curses_addstr(s)
        value s;
{
  addstr(String_val(s));
  return Val_unit;
}

/* L'exemple continue sur plusieurs pages. */
```

(En fait, il serait plus astucieux de créer une bibliothèque pour ce code enveloppe, avec chaque fonction enveloppe dans un fichier séparé, pour que l'éditeur de liens puisse extraire de la bibliothèque `curses` les seules fonctions effectivement utilisées par le programme Caml Light.)

Le fichier `curses.c` peut être compilé avec le compilateur C :

```
cc -c -I/usr/local/lib/caml-light curses.c
```

ou, plus simplement, avec le compilateur de Caml Light

```
camlc -c curses.c
```

(Quand on lui passe un fichier `.c`, la commande `camlc` appelle tout simplement `cc` sur ce fichier, avec la bonne option `-I`.)

Et maintenant, voici un exemple simple de programme Caml Light qui utilise le module `curses` :

```
#open "curses";;
let main_window = initscr () in
let small_window = newwin 10 5 20 10 in
  mvwaddstr main_window 10 2 "Hello";
```

```
  mvwaddstr small_window 4 3 "world";
  refresh();
  for i = 1 to 100000 do () done;
  endwin();;
```

Pour compiler ce programme (`test.ml`), lancez :

```
camlc -c test.ml
```

Finalement, il reste à faire l'édition de liens du tout :

```
camlc -custom -o test test.zo curses.o -lcurses
```

III

La bibliothèque standard du système Caml Light

9
La bibliothèque de base

Ce chapitre décrit les modules de la bibliothèque de base de Caml Light. Cette bibliothèque joue un rôle particulier dans le système, pour deux raisons :

- Elle est automatiquement liée avec les fichiers de code objet produits par la commande `camlc` (chapitre 3). Les fonctions définies dans la bibliothèque de base peuvent donc être utilisées dans les programmes indépendants sans qu'il soit nécessaire d'ajouter des fichiers `.zo` à la ligne de commande de la phase d'édition de liens. De même, en usage interactif, ces fonctions peuvent être utilisées directement dans les phrases entrées par l'utilisateur, sans que l'on ait à charger au préalable un fichier de code.
- Les interfaces de certains modules de cette bibliothèque sont automatiquement "ouvertes" au début de chaque compilation et au lancement du système interactif. Il est donc possible de se servir d'identificateurs non qualifiés pour faire référence aux fonctions fournies par ces modules, sans ajouter de directives `#open`. La liste des modules ouverts automatiquement dépend de l'option `-O` donnée au compilateur ou au système interactif :

Option `-O`	Modules ouverts (dans l'ordre dans lequel ils sont parcourus pendant la complétion)
`-O cautious` (cas par défaut)	`io`, `eq`, `int`, `float`, `ref`, `pair`, `list`, `vect`, `char`, `string`, `bool`, `exc`, `stream`, `builtin`
`-O fast`	`io`, `eq`, `int`, `float`, `ref`, `pair`, `list`, `fvect`, `fchar`, `fstring`, `bool`, `exc`, `stream`, `builtin`
`-O none`	`builtin`

Conventions

Pour faciliter la recherche, les modules sont présentés par ordre alphabétique. Pour chaque module, les déclarations de son fichier d'interface sont imprimées une par une en police "machine à écrire", suivie chacune par un court commentaire. Tous les modules et les identificateurs qu'ils exportent sont indexés à la fin de cet ouvrage.

9.1 `bool` : opérations sur les booléns

```
value prefix not : bool -> bool
```

La négation booléenne.

9.2 `builtin` : types et constructeurs de base

Ce module définit des types et des exceptions correspondant à des constructions syntaxiques du langage qui jouent donc un rôle particulier dans le compilateur.

```
type int
type float
type string
type char
```

Les types des nombres entiers, des nombres en virgule flottante, des chaînes de caractères et des caractères, respectivement.

```
type exn
```

Le type des valeurs exceptionnelles.

```
type bool = false | true
```

Le type des valeurs booléennes.

```
type 'a vect
```

Le type des tableaux dont les éléments sont de type `'a`.

```
type unit = ()
```

Le type de la valeur « rien ».

```
type 'a list = [] | prefix :: of 'a * 'a list
```

Le type des listes.

```
exception Match_failure of string * int * int
```

L'exception déclenchée quand un filtrage échoue. L'argument du constructeur indique la position dans le texte source du filtrage qui a échoué (nom du fichier source, position du premier caractère du filtrage, position du dernier caractère du filtrage).

9.3 `char` : opérations sur les caractères

```
value int_of_char : char -> int
```

Renvoie le code ASCII de son argument.

```
value char_of_int : int -> char
```

Renvoie le caractère de code ASCII donné. Déclenche `Invalid_argument "char_of_int"` si l'argument est en dehors de l'intervalle 0–255.

`value char_for_read : char -> string`

Renvoie une chaîne représentant le caractère donné, avec les caractères spéciaux codés par des séquences d'échappement, suivant les mêmes conventions que Caml Light.

9.4 `eq` : fonctions d'égalité

`value prefix = : 'a -> 'a -> bool`

`e1 = e2` teste l'égalité structurelle de `e1` et `e2`. Les structures mutables (par exemple les références) sont égales si et seulement si leurs contenus courants sont structurellement égaux, même si les deux objets mutables ne sont pas le même objet physique. L'égalité entre valeurs fonctionnelles déclenche l'exception `Invalid_argument`. L'égalité entre structures de données cycliques peut ne pas terminer.

`value prefix <> : 'a -> 'a -> bool`

Négation de `prefix =`.

`value prefix == : 'a -> 'a -> bool`

`e1 == e2` teste l'égalité physique de `e1` et `e2`. Sur les entiers et les caractères, c'est équivalent de l'égalité structurelle. Sur les structures mutables, `e1 == e2` est vrai si et seulement si une modification physique de `e1` affecte aussi `e2`. Sur les structures non mutables, le comportement de `prefix ==` dépend de l'implémentation, mais `e1 == e2` implique toujours `e1 = e2`.

`value prefix != : 'a -> 'a -> bool`

Négation de `prefix ==`.

9.5 `exc` : exceptions

`value raise : exn -> 'a`

Déclenche une valeur exceptionnelle.

Exceptions prédéfinies d'usage général

`exception Out_of_memory`

Déclenchée par le récupérateur de mémoire (en anglais *garbage collector*) quand la mémoire disponible est insuffisante pour terminer le calcul.

`exception Invalid_argument of string`

Déclenchée par certaines fonctions de bibliothèque pour signaler que les arguments donnés n'ont pas de sens.

`exception Failure of string`

Déclenchée par certaines fonctions de bibliothèque pour signaler qu'elles ne sont pas définies sur les argument donnés.

```
exception Not_found
```

Déclenchée par certaines fonctions de recherche quand l'objet cherché est introuvable.

```
exception Exit
```

Cette exception n'est pas déclenchée par les fonctions de bibliothèque. Elle vous est fournie pour que vous l'utilisiez dans vos programmes.

```
value failwith : string -> 'a
```

Déclenche l'exception `Failure` avec pour argument la chaîne donnée.

```
value invalid_arg : string -> 'a
```

Déclenche l'exception `Invalid_argument` avec pour argument la chaîne donnée.

9.6 `fchar` : opérations sur les caractères, sans vérifications

Ce module fournit les mêmes fonctions que le module `char` (section 9.3), mais ne fait pas de tests de bornes sur les arguments des fonctions. Les fonctions sont donc plus rapides que celles du module `char`, mais le programme peut faire n'importe quoi si l'on appelle une de ces fonctions avec des paramètres incorrects, c'est-à-dire des paramètres qui auraient déclenché l'exception `Invalid_argument` lors de l'appel de la fonction correspondante du module `char`.

9.7 `float` : opérations sur les nombres flottants

Les caractéristiques des nombres flottants (précision, intervalle de valeurs représentables, mode d'arrondi, ...) dépendent de l'implémentation et de la machine. Autant que possible, l'implémentation essaye d'utiliser des flottants double précision à la norme IEEE. Le comportement des opérations flottantes en cas de débordement est non spécifié.

```
value int_of_float : float -> int
```

Convertit un flottant en un entier. Si le flottant est positif ou nul, il est tronqué à l'entier immédiatement inférieur ou égal. Le résultat n'est pas spécifié si l'argument est négatif, ou si l'argument tombe en dehors de l'intervalle des entiers représentables.

```
value float_of_int : int -> float
```

Convertit un entier en flottant.

```
value minus : float -> float
value minus_float : float -> float
```

Opposé.

```
value prefix + : float -> float -> float
value prefix +. : float -> float -> float
value add_float : float -> float -> float
```

Addition.

```
value prefix - : float -> float -> float
value prefix -. : float -> float -> float
value sub_float : float -> float -> float
```

Soustraction.

```
value prefix * : float -> float -> float
value prefix *. : float -> float -> float
value mult_float : float -> float -> float
```

Multiplication.

```
value prefix / : float -> float -> float
value prefix /. : float -> float -> float
value div_float : float -> float -> float
```

Division. Comportement non spécifié si le dividende est 0.0.

```
value eq_float : float -> float -> bool
value prefix =. : float -> float -> bool
```

Égalité flottante. Équivalente à l'égalité générique, mais un peu plus efficace.

```
value neq_float : float -> float -> bool
value prefix <>. : float -> float -> bool
```

Négation d'`eq_float`.

```
value prefix < : float -> float -> bool
value prefix <. : float -> float -> bool
value lt_float : float -> float -> bool
value prefix > : float -> float -> bool
value prefix >. : float -> float -> bool
value gt_float : float -> float -> bool
value prefix <= : float -> float -> bool
value prefix <=. : float -> float -> bool
value le_float : float -> float -> bool
value prefix >= : float -> float -> bool
value prefix >=. : float -> float -> bool
value ge_float : float -> float -> bool
```

Comparaisons entre flottants.

```
value exp : float -> float
value log : float -> float
value sqrt : float -> float
value power : float -> float -> float
value sin : float -> float
value cos : float -> float
value tan : float -> float
```

```
value asin : float -> float
value acos : float -> float
value atan : float -> float
value atan2 : float -> float -> float
```

Fonctions transcendantes bien connues sur les flottants.

```
value abs_float : float -> float
```

Renvoie la valeur absolue de son argument.

```
value string_of_float : float -> string
```

Convertit un flottant en sa représentation décimale.

```
value float_of_string : string -> float
```

Convertit une chaîne, représentant un flottant en décimal, en un flottant. Le résultat n'est pas spécifié si la chaîne donnée n'est pas la représentation valide d'un flottant.

9.8 `fstring` : opérations sur les chaînes de caractères, sans vérifications

Ce module fournit les mêmes fonctions que le module `string` (section 9.16), mais ne fait pas de tests de bornes sur les arguments des fonctions. Les fonctions sont donc plus rapides que celles du module `string`, mais le programme peut faire n'importe quoi si l'on appelle une de ces fonction avec des paramètres incorrects, c'est-à-dire des paramètres qui auraient déclenché l'exception `Invalid_argument` lors de l'appel de la fonction correspondante du module `string`.

9.9 `fvect` : opérations sur les tableaux, sans vérifications

Ce module fournit les mêmes fonctions que le module `vect` (section 9.17), mais ne fait pas de tests de bornes sur les arguments des fonctions. Les fonctions sont donc plus rapides que celles du module `vect`, mais le programme peut faire n'importe quoi si l'on appelle une de ces fonction avec des paramètres incorrects, c'est-à-dire des paramètres qui auraient déclenché l'exception `Invalid_argument` lors de l'appel de la fonction correspondante du module `vect`.

9.10 `int` : opérations sur les entiers

Les entiers sont représentés sur 31 bits. Toutes les opérations sont prises modulo 2^{31}. Elles n'échouent pas en cas de débordement.

```
exception Division_by_zero
```

Déclenchée quand on divise par zéro ou quand on calcule un modulo avec zéro.

```
value minus : int -> int
value minus_int : int -> int
```

Opposé. On peut écrire `-e` au lieu de `minus e`.

```
value succ : int -> int
```

`succ x` vaut `x+1`.

```
value pred : int -> int
```

`pred x` vaut `x-1`.

```
value prefix + : int -> int -> int
value add_int : int -> int -> int
```

Addition.

```
value prefix - : int -> int -> int
value sub_int : int -> int -> int
```

Soustraction.

```
value prefix * : int -> int -> int
value mult_int : int -> int -> int
```

Multiplication.

```
value prefix / : int -> int -> int
value div_int : int -> int -> int
value prefix quo : int -> int -> int
```

Division entière. Déclenche `Division_by_zero` si le second argument est 0. Résultat non spécifié si l'un des arguments est négatif.

```
value prefix mod : int -> int -> int
```

Reste. Déclenche `Division_by_zero` si le second argument est 0. Résultat non spécifié si l'un des arguments est négatif.

```
value eq_int : int -> int -> bool
```

Égalité entière. Équivalente à l'égalité générique, mais plus efficace.

```
value neq_int : int -> int -> bool
```

Négation d'`eq_int`.

```
value prefix < : int -> int -> bool
value lt_int : int -> int -> bool
value prefix > : int -> int -> bool
value gt_int : int -> int -> bool
value prefix <= : int -> int -> bool
value le_int : int -> int -> bool
value prefix >= : int -> int -> bool
value ge_int : int -> int -> bool
```

Comparaisons entre entiers.

```
value min : int -> int -> int
```

Renvoie le plus petit de ses arguments.

`value max : int -> int -> int`

Renvoie le plus grand de ses arguments.

`value abs : int -> int`

Renvoie la valeur absolue de son argument.

Opérations bit à bit

`value prefix land : int -> int -> int`

« Et » logique bit à bit.

`value prefix lor : int -> int -> int`

« Ou » logique bit à bit.

`value prefix lxor : int -> int -> int`

« Ou exclusif » logique bit à bit.

`value lnot : int -> int`

Complément logique bit à bit

```
value prefix lsl : int -> int -> int
value lshift_left : int -> int -> int
```

`n lsl m`, ou de façon équivalente `lshift_left n m`, décale `n` vers la gauche de `m` bits.

`value prefix lsr : int -> int -> int`

`n lsr m` décale `n` vers la droite de `m` bits. C'est un décalage logique : des bits à zéro sont introduits à gauche, sans tenir compte du signe.

```
value prefix asr : int -> int -> int
value lshift_right : int -> int -> int
```

`n asr m`, ou de façon équivalente `lshift_right n m`, décale `n` vers la droite de `m` bits. C'est un décalage arithmétique : le bit de signe est recopié.

Fonctions de conversion

`value string_of_int : int -> string`

Convertit un entier en une chaîne de caractères, en représentation décimale.

`value int_of_string : string -> int`

Convertit une chaîne en un entier, en décimal (cas par défaut) ou en hexadécimal, octal, ou binaire si la chaîne commence par `0x`, `0o` ou `0b`. Déclenche `Failure "int_of_string"` si la chaîne donnée n'est pas la représentation d'un entier.

9.11 `io` : entrées-sorties avec tampons

```
type in_channel
type out_channel
```

Les types abstraits des canaux (en anglais *channel*) d'entrées-sorties.

```
exception End_of_file
```

Déclenchée quand une opération de lecture ne peut être achevée parce que la fin du fichier est atteinte.

```
value stdin : in_channel
value std_in : in_channel
value stdout : out_channel
value std_out : out_channel
value stderr : out_channel
value std_err : out_channel
```

Les canaux standard du processus : entrée standard (standard input), sortie standard (standard output) et sortie d'erreur standard (standard error). `std_in`, `std_out` et `std_err` sont respectivement synonymes de `stdin`, `stdout` et `stderr`.

```
value exit : int -> 'a
```

Vide les tampons associés à `std_out` et `std_err` et termine le processus, en renvoyant son argument comme code de retour (en anglais *status code*) au système d'exploitation. Par convention, le code 0 indique qu'il n'y a pas eu d'erreurs, et un petit entier positif signale un échec. Cette fonction doit être appelée à la fin de tous les programmes indépendants qui écrivent des résultats sur `std_out` ou `std_err` ; à défaut, le programme peut sembler ne pas produire de résultats, ou ses sorties peuvent être tronquées.

Fonctions de sorties sur la sortie standard

```
value print_char : char -> unit
```

Imprime un caractère sur la sortie standard.

```
value print_string : string -> unit
```

Imprime une chaîne sur la sortie standard.

```
value print_int : int -> unit
```

Imprime un entier, en décimal, sur la sortie standard.

```
value print_float : float -> unit
```

Imprime un flottant, en décimal, sur la sortie standard.

```
value print_endline : string -> unit
```

Imprime une chaîne, suivie par un caractère saut de ligne, sur la sortie standard.

```
value print_newline : unit -> unit
```

Imprime un caractère saut de ligne sur la sortie standard et vide cette dernière.

Fonctions de sorties sur la sortie standard error

`value prerr_char : char -> unit`

Imprime un caractère sur la sortie d'erreur standard.

`value prerr_string : string -> unit`

Imprime une chaîne sur la sortie d'erreur standard.

`value prerr_int : int -> unit`

Imprime un entier, en décimal, sur la sortie d'erreur standard.

`value prerr_float : float -> unit`

Imprime un flottant, en décimal, sur la sortie d'erreur standard.

`value prerr_endline : string -> unit`

Imprime une chaîne, suivie d'un caractère saut de ligne, sur la sortie d'erreur standard et vide cette dernière.

Fonctions d'entrée sur l'entrée standard

`value read_line : unit -> string`

Vide la sortie standard, puis lit des caractères sur l'entrée standard jusqu'à rencontrer un caractère saut de ligne. Renvoie la chaîne de tous les caractères lus, sans le caractère saut de ligne final.

`value read_int : unit -> int`

Vide la sortie standard, puis lit une ligne sur l'entrée standard et la convertit en un entier. Déclenche `Failure "int_of_string"` si la ligne lue n'est pas la représentation d'un entier.

`value read_float : unit -> float`

Vide la sortie standard, puis lit une ligne sur l'entrée standard et la convertit en un flottant. Le résultat n'est pas spécifié si la ligne lue n'est pas la représentation d'un flottant.

Fonctions générales de sortie

`value open_out : string -> out_channel`

Ouvre le fichier de nom donné en écriture et renvoie un nouveau canal de sortie sur ce fichier, positionné au début du fichier. Le fichier est tronqué à la taille zéro s'il existe déjà. Il est créé s'il n'existait pas. Déclenche `sys__Sys_error` si le fichier ne peut être ouvert.

`value open_out_bin : string -> out_channel`

Analogue à `open_out`, mais le fichier est ouvert en mode binaire, et non pas en mode texte, ce qui inhibe toute conversion pendant les écritures. Sur les systèmes d'exploitation qui ne distinguent pas le mode texte du mode binaire, cette fonction se comporte comme `open_out`.

`value open_out_gen :`
`sys__open_flag list -> int -> string -> out_channel`

`open_out_gen mode droits nom` ouvre le fichier de nom `nom` en écriture. L'argument `mode` spécifie le mode d'ouverture (voir `sys__open`). L'argument `droits` spécifie les permissions attribuées au fichier, au cas où il est nécessaire de le créer (voir `sys__open`). `open_out` et `open_out_bin` sont des cas particuliers de cette fonction.

`value open_descriptor_out : int -> out_channel`

`open_descriptor_out df` renvoie un canal de sortie qui écrit sur le descripteur de fichier `df`. Le descripteur de fichier `df` doit avoir été préalablement ouvert en écriture ; à défaut, le comportement de cette fonction n'est pas spécifié.

`value flush : out_channel -> unit`

Vide le tampon associé au canal de sortie donné, en achevant toutes les écritures en instance sur ce canal. Les programmes interactifs doivent prendre soin de vider `std_out` aux bons moments.

`value output_char : out_channel -> char -> unit`

Écrit un caractère sur le canal de sortie donné.

`value output_string : out_channel -> string -> unit`

Écrit une chaîne sur le canal de sortie donné.

`value output : out_channel -> string -> int -> int -> unit`

`output can tamp index longueur` écrit `longueur` caractères de la chaîne `tamp`, à partir de la position `index`, sur le canal de sortie `can`. Déclenche `Invalid_argument "output"` si `index` et `longueur` ne désignent pas une sous-chaîne valide de `tamp`.

`value output_byte : out_channel -> int -> unit`

Écrit un entier sur huit bits sous la forme du caractère de ce code sur le canal de sortie donné. L'entier donné est pris modulo 256.

`value output_binary_int : out_channel -> int -> unit`

Écrit une représentation binaire d'un entier sur le canal de sortie donné. La seule façon sûre de le relire est d'utiliser la fonction `input_binary_int`. La représentation est compatible entre toutes les machines pour une version donnée de Caml Light.

`value output_value : out_channel -> 'a -> unit`

Écrit la représentation d'une valeur structurée d'un type quelconque (sauf les fonctions) sur le canal de sortie donné. Les circularités et le partage à l'intérieur de la valeur sont détectés et préservés. L'objet peut être relu par la fonction `input_value`. La représentation est compatible entre toutes les machines pour une version donnée de Caml Light. Déclenche `Invalid_argument "extern: functional value"` quand l'argument contient une valeur fonctionnelle.

`value seek_out : out_channel -> int -> unit`

`seek_out can pos` fixe la position courante d'écriture du canal `can` à `pos`. Fonctionne uniquement pour les fichiers normaux. Pour les autres types de fichiers (terminaux, tuyaux (en anglais *pipes*) et prises (en anglais *sockets*)), le comportement n'est pas spécifié.

`value pos_out : out_channel -> int`

Renvoie la position d'écriture courante du canal donné.

`value out_channel_length : out_channel -> int`

Renvoie la taille totale (nombre de caractères) du canal donné.

`value close_out : out_channel -> unit`

Ferme un canal, après avoir vidé le tampon associé. Le programme peut faire n'importe quoi si l'une quelconque des fonctions ci-dessus est appelée sur un canal fermé.

Fonctions générales d'entrée

`value open_in : string -> in_channel`

Ouvre le fichier de nom donné en lecture et renvoie un nouveau canal d'entrée lisant sur ce fichier, positionné au début du fichier. Déclenche `sys__Sys_error` si le fichier n'a pas pu être ouvert.

`value open_in_bin : string -> in_channel`

Analogue à `open_in`, mais le fichier est ouvert en mode binaire, et non pas en mode texte, ce qui inhibe toute conversion pendant les lectures. Sur les systèmes d'exploitation qui ne distinguent pas le mode texte du mode binaire, cette fonction se comporte comme `open_in`.

```
value open_in_gen :
   sys__open_flag list -> int -> string -> in_channel
```

`open_in_gen mode droits nom` ouvre en lecture le fichier de nom `nom`. Les arguments `mode` et `droits` spécifient le mode d'ouverture et les permissions du fichier (voir `sys__open`). Les fonctions `open_in` et `open_in_bin` sont des cas particuliers de cette fonction.

`value open_descriptor_in : int -> in_channel`

`open_descriptor_in df` renvoie un canal d'entrée qui lit sur le descripteur de fichier `df`. Le descripteur de fichier `df` doit avoir été préalablement ouvert en lecture ; à défaut, le comportement de cette fonction n'est pas spécifié.

`value input_char : in_channel -> char`

Lit un caractère sur un canal d'entrée. Déclenche `End_of_file` si la fin du fichier est atteinte.

`value input_line : in_channel -> string`

Lit des caractères sur le canal d'entrée donné, jusqu'à rencontrer un caractère saut de ligne. Renvoie la chaîne de tous les caractères lus, sans le caractère saut de ligne à la fin. Déclenche `End_of_file` si la fin du fichier est atteinte avant que la ligne ne soit complète.

`value input : in_channel -> string -> int -> int -> int`

`input can tamp index longueur` tente de lire `longueur` caractères sur le canal `can`, en les stockant dans la chaîne `tamp`, à partir du caractère numéro `index`. La fonction renvoie le nombre de caractères lus, entre 0 et `longueur` (compris). Le résultat 0 signifie que la fin du fichier a été atteinte. Un résultat entre 0 et `longueur` (non compris) signifie qu'il n'y a plus de caractères disponibles actuellement ; `input` doit être appelée à nouveau pour lire les caractères suivants, si nécessaire. L'exception `Invalid_argument "input"` est déclenchée si `index` et `longueur` ne désignent pas une sous-chaîne valide de `tamp`.

`value really_input : in_channel -> string -> int -> int -> unit`

`really_input can tamp index longueur` lit `longueur` caractères sur le canal `can` et les copie dans la chaîne `tamp`, à partir du caractère numéro `index`. Contrairement à `input`, `really_input` fait plusieurs tentatives de lecture si nécessaire pour lire exactement le nombre de caractères demandés. Déclenche `End_of_file` si la fin du fichier est atteinte avant que `longueur` caractères ne soient lus. Déclenche `Invalid_argument "really_input"` si `index` et `longueur` ne désignent pas une sous-chaîne valide de `tamp`.

`value input_byte : in_channel -> int`

Analogue à `input_char`, mais renvoie l'entier sur huit bits qui représente le caractère lu. Déclenche `End_of_file` si la fin du fichier est atteinte.

`value input_binary_int : in_channel -> int`

Lit un entier codé en représentation binaire sur le canal d'entrée donné. (Voir la fonction `output_binary_int` page 103.) Déclenche `End_of_file` si la fin du fichier est atteinte pendant la lecture de l'entier.

`value input_value : in_channel -> 'a`

Lit la représentation d'une valeur structurée, produite par `output_value`, et renvoie la valeur correspondante. Déclenche `End_of_file` si la fin du fichier est atteinte. Déclenche `Failure` avec un message descriptif si les caractères lus dans le fichier ne représentent pas une valeur structurée. Ce mécanisme n'est pas typé de manière sûre. Le type de l'objet retourné n'est pas `'a` à proprement parler. L'objet retourné possède non pas tous les types, mais un certain type qui ne peut pas être déterminé à la compilation. Le programmeur est donc responsable du bon typage et devrait toujours donner explicitement le type attendu de la valeur retournée, à l'aide d'une contrainte de type : `(input_value canal : type)`. Le programme peut faire n'importe quoi si l'objet lu dans le fichier n'appartient pas au type donné.

`value seek_in : in_channel -> int -> unit`

`seek_in can pos` fixe la position de lecture courante du canal `can` à `pos`. Fonctionne uniquement pour les fichiers normaux.

`value pos_in : in_channel -> int`

Renvoie la position de lecture courante du canal donné.

`value in_channel_length : in_channel -> int`

Renvoie la taille totale (nombre de caractères) du canal donné. Fonctionne uniquement pour les fichiers normaux. Pour les autres fichiers, le résultat est non spécifié.

`value close_in : in_channel -> unit`

Ferme le canal donné. Le programme peut faire n'importe quoi si l'une quelconque des fonctions ci-dessus est appelée sur un canal fermé.

9.12 `list` : opérations sur les listes

`value list_length : 'a list -> int`

Renvoie la longueur (nombre d'éléments) de la liste donnée.

`value prefix @ : 'a list -> 'a list -> 'a list`

Concaténation de deux listes.

`value hd : 'a list -> 'a`

Renvoie le premier élément de la liste donnée. Déclenche `Failure "hd"` si la liste est vide.

`value tl : 'a list -> 'a list`

Renvoie la liste donnée sans son premier élément. Déclenche `Failure "tl"` si la liste est vide.

`value rev : 'a list -> 'a list`

Renverse une liste.

`value map : ('a -> 'b) -> 'a list -> 'b list`

`map f [a1; ...; an]` applique tour à tour la fonction `f` à `a1` ... `an` et construit la liste `[f a1; ...; f an]` des résultats retournés par `f`.

`value do_list : ('a -> 'b) -> 'a list -> unit`

`do_list f [a1; ...; an]` applique tour à tour la fonction `f` à `a1` ... `an`, ignorant tous les résultats. Équivaut à `begin f a1; f a2; ...; f an; () end`.

`value it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

`it_list f a [b1; ...; bn]` vaut `f (... (f (f a b1) b2) ...) bn`.

`value list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

`list_it f [a1; ...; an] b` vaut `f a1 (f a2 (... (f an b) ...))`.

`value map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`

`map2 f [a1;...; an] [b1;...; bn]` vaut `[f a1 b1;...; f an bn]`. Déclenche `Invalid_argument "map2"` si les deux listes n'ont pas la même longueur.

`value do_list2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> unit`

`do_list2 f [a1; ...; an] [b1; ...; bn]` applique tour à tour `f a1 b1; ...; f an bn`, ignorant tous les résultats. Déclenche `Invalid_argument "do_list2"` si les deux listes n'ont pas la même longueur.

```
value it_list2 :
    ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
```

`it_list2 f a [b1; ...; bn] [c1; ...; cn]` vaut
`f (... (f (f a b1 c1) b2 c2) ...) bn cn`.
Déclenche `Invalid_argument "it_list2"` si les deux listes n'ont pas la même longueur.

```
value list_it2 :
    ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
```

`list_it2 f [a1; ...; an] [b1; ...; bn] c` vaut
`f a1 b1 (f a2 b2 (... (f an bn c) ...))`.
Déclenche `Invalid_argument "list_it2"` si les deux listes n'ont pas la même longueur.

`value flat_map : ('a -> 'b list) -> 'a list -> 'b list`

`flat_map f [l1; ...; ln]` vaut `(f l1) @ (f l2) @ ... @ (f ln)`.

`value for_all : ('a -> bool) -> 'a list -> bool`

`for_all p [a1; ...; an]` vaut `(p a1) & (p a2) & ... & (p an)`.

`value exists : ('a -> bool) -> 'a list -> bool`

`exists p [a1; ...; an]` vaut `(p a1) or (p a2) or ... or (p an)`.

`value mem : 'a -> 'a list -> bool`

`mem a l` est vrai si et seulement si `a` est structurellement égal (au sens de la fonction `=` du module `eq`, section 9.4) à un élément de `l`.

`value memq : 'a -> 'a list -> bool`

`memq a l` est vrai si et seulement si `a` est physiquement égal (au sens de la fonction `==` du module `eq`, section 9.4) à un élément de `l`.

`value except : 'a -> 'a list -> 'a list`

`except a l` renvoie la liste `l` privée du premier élément structurellement égal à `a`. La liste `l` est renvoyée sans changement si elle ne contient pas `a`.

`value exceptq : 'a -> 'a list -> 'a list`

Même comportement que `except`, mais compare les éléments avec l'égalité physique au lieu de l'égalité structurelle.

`value subtract : 'a list -> 'a list -> 'a list`

`subtract l1 l2` renvoie la liste `l1` privée de tous les éléments structurellement égaux à l'un des éléments de `l2`.

`value union : 'a list -> 'a list -> 'a list`

`union l1 l2` place devant la liste `l2` tous les éléments de la liste `l1` qui ne sont pas structurellement égaux à l'un des éléments de `l2`.

`value intersect : 'a list -> 'a list -> 'a list`

`intersect l1 l2` renvoie la liste des éléments de `l1` qui sont structurellement égaux à l'un des éléments de `l2`.

`value index : 'a -> 'a list -> int`

`index a l` renvoie la position du premier élément de la liste `l` qui est structurellement égal à `a`. La tête de la liste a la position 0. Déclenche `Not_found` si `a` n'est pas présent dans `l`.

`value assoc : 'a -> ('a * 'b) list -> 'b`

`assoc a l` renvoie la valeur associée à la clef `a` dans la liste de paires `l`. C'est-à-dire que `assoc a [ ...; (a,b); ...]` est égal à `b` si `(a,b)` est la paire la plus à gauche dans `l` ayant `a` comme première composante. Déclenche `Not_found` s'il n'y a pas de valeur associée à `a` dans la liste `l`.

`value assq :  'a -> ('a * 'b) list -> 'b`

Même comportement que `assoc`, mais compare les clefs avec l'égalité physique au lieu de l'égalité structurelle.

`value mem_assoc : 'a -> ('a * 'b) list -> bool`

Similaire à `assoc`, mais renvoie simplement `true` s'il existe une liaison pour la clef donnée, et `false` sinon.

9.13 pair : opérations sur les paires

`value fst : 'a * 'b -> 'a`

Renvoie la première composante d'une paire.

`value snd : 'a * 'b -> 'b`

Renvoie la seconde composante d'une paire.

`value split : ('a * 'b) list -> 'a list * 'b list`

Transforme une liste de paires en une paire de listes :
`split [(a1,b1);...; (an,bn)]` vaut `([a1;...; an], [b1;...; bn])`

`value combine : 'a list * 'b list -> ('a * 'b) list`

Transforme une paire de listes en une liste de paires :
`combine ([a1;...; an], [b1;...; bn])` vaut
`[(a1,b1);...; (an,bn)]`. Déclenche `Invalid_argument "combine"` si les deux listes n'ont pas la même longueur.

```
value map_combine :
        ('a * 'b -> 'c) -> 'a list * 'b list -> 'c list
```

`map_combine f ([a1; ...; an], [b1; ...; bn])` vaut `[f (a1, b1);...; f (an, bn)]`. Déclenche `invalid_argument "map_combine"` si les deux listes n'ont pas la même longueur.

```
value do_list_combine :
        ('a * 'b -> 'c) -> 'a list * 'b list -> unit
```

`do_list_combine f ([a1; ...; an], [b1; ...; bn])` effectue `f (a1, b1); ...; f (an, bn)`, ignorant tous les résultats. Déclenche `Invalid_argument "do_list_combine"` si les deux listes n'ont pas la même longueur.

9.14 `ref` : opérations sur les références

`type 'a ref = ref of mutable 'a`

Le type des références vers une valeur de type `'a`.

`value prefix ! : 'a ref -> 'a`

`!r` renvoie le contenu actuel de la référence `r`. Pourrait être défini par `fun (ref x) -> x`.

`value prefix := : 'a ref -> 'a -> unit`

`r := a` écrit la valeur de `a` dans la référence `r`.

`value incr : int ref -> unit`

Incrémente l'entier contenu dans une référence. Pourrait être défini par `fun r -> r := succ !r`.

`value decr : int ref -> unit`

Décrémente l'entier contenu dans une référence. Pourrait être défini par `fun r -> r := pred !r`.

9.15 `stream` : opérations sur les flux

`type 'a stream`

Le type des flux (en anglais *streams*) dont les éléments ont le type `'a`.

`exception Parse_failure`

Déclenchée par les analyseurs syntaxiques quand aucun des premiers composants des motifs de l'analyseur ne filtre le flux.

`exception Parse_error`

Déclenchée par les analyseurs syntaxiques quand un des motifs de l'analyseur a été sélectionné au vu de son premier composant, mais que le reste du motif ne filtre pas le reste du flux.

`value stream_next : 'a stream -> 'a`

`stream_next s` renvoie le premier élément du flux `s` et le retire du flux. Déclenche `Parse_failure` si le flux est vide.

`value stream_from : (unit -> 'a) -> 'a stream`

`stream_from f` renvoie le flux qui fabrique ses éléments en appelant la fonction `f`. Cette fonction pourrait être définie par :

```
let rec stream_from f = [< 'f(); stream_from f >]
```

mais est implémentée plus efficacement, en utilisant des opérations internes de bas niveau sur les flux.

`value stream_of_string : string -> char stream`

`stream_of_string s` renvoie le flux des caractères de la chaîne `s`.

`value stream_of_channel : in_channel -> char stream`

`stream_of_channel ic` renvoie le flux des caractères lus sur le canal `ic`.

`value do_stream : ('a -> 'b) -> 'a stream -> unit`

`do_stream f s` parcourt tout le flux `s`, en appliquant tour à tour la fonction `f` à chaque élément rencontré. Le flux `s` est vidé.

`value stream_check : ('a -> bool) -> 'a stream -> 'a`

`stream_check p` renvoie l'analyseur syntaxique qui retourne le premier élément du flux si cet élément vérifie le prédicat `p`, et déclenche `Parse_failure` sinon.

`value end_of_stream : 'a stream -> unit`

Renvoie `()` si le flux est vide, et déclenche `Parse_failure` sinon.

`value stream_get : 'a stream -> 'a * 'a stream`

`stream_get s` renvoie le premier élément du flux `s` et un flux contenant les autres éléments de `s`. Déclenche `Parse_failure` si le flux est vide. Le flux `s` n'est pas modifié. Cette fonction permet l'accès non destructif à un flux.

9.16 `string` : opérations sur les chaînes de caractères

`value string_length : string -> int`

Renvoie la longueur (nombre de caractères) de la chaîne donnée.

`value nth_char : string -> int -> char`

`nth_char s n` renvoie le caractère numéro `n` de la chaîne `s`. Le premier caractère est le caractère numéro 0. Le dernier caractère a le numéro `string_length s - 1`. Déclenche `Invalid_argument "nth_char"` si `n` est en dehors de l'intervalle 0, `(string_length s - 1)`.

`value set_nth_char : string -> int -> char -> unit`

`set_nth_char s n c` modifie la chaîne `s` en place, remplaçant le caractère numéro `n` par `c`. Déclenche `Invalid_argument "set_nth_char"` si `n` est en dehors de l'intervalle 0, `(string_length s - 1)`.

`value prefix ^ : string -> string -> string`

`s1 ^ s2` renvoie une nouvelle chaîne contenant la concaténation des chaînes `s1` et `s2`.

`value sub_string : string -> int -> int -> string`

`sub_string s debut long` renvoie une nouvelle chaîne de longueur `long`, contenant les caractères numéro `debut` jusqu'à `debut + long - 1` de la chaîne `s`. Déclenche `Invalid_argument "sub_string"` si `debut` et `long` ne désignent pas une sous-chaîne valide de `s`, c'est-à-dire si `debut < 0`, ou `long < 0`, ou `debut + long > string_length s`.

`value create_string : int -> string`

`create_string n` renvoie une nouvelle chaîne de longueur `n`. La chaîne contient initialement des caractères arbitraires.

`value make_string : int -> char -> string`

`make_string n c` renvoie une nouvelle chaîne de longueur `n`, remplie avec le caractère `c`.

`value fill_string : string -> int -> int -> char -> unit`

`fill_string s debut long c` modifie physiquement la chaîne `s`, en remplaçant les caractères numéro `debut` à `debut + long - 1` par `c`. Déclenche `Invalid_argument "fill_string"` si `debut` et `long` ne désignent pas une sous-chaîne valide de `s`.

`value blit_string: string -> int -> string -> int -> int -> unit`

`blit_string s1 o1 s2 o2 long` copie `long` caractères de la chaîne `s1`, à partir du caractère numéro `o1`, dans la chaîne `s2`, à partir du caractère numéro `o2`. Cette fonction marche correctement même si `s1` et `s2` désignent la même chaîne et que les sous-chaînes source et destination se recouvrent. Déclenche `Invalid_argument "blit_string"` si `o1` et `long` ne désignent pas une sous-chaîne valide de `s1`, ou si `o2` et `long` ne désignent pas une sous-chaîne valide de `s2`.

`value replace_string : string -> string -> int -> unit`

`replace_string dest src debut` copie tous les caractères de la chaîne `src` dans la chaîne `dest`, à partir du caractère numéro `debut` de `dest`. Déclenche `Invalid_argument "replace_string"` si la copie ne tient pas dans la chaîne `dest`.

```
value eq_string : string -> string -> bool
value neq_string : string -> string -> bool
value le_string : string -> string -> bool
value lt_string : string -> string -> bool
value ge_string : string -> string -> bool
value gt_string : string -> string -> bool
```

Fonctions de comparaison (par ordre lexicographique) entre chaînes.

`value compare_strings : string -> string -> int`

Comparaison générale entre chaînes. `compare_strings s1 s2` renvoie `0` si `s1` et `s2` sont égales, `-2` si `s1` est un préfixe de `s2`, `2` si `s2` est un préfixe de `s1` ; sinon elle renvoie `-1` si `s1` est avant `s2` dans l'ordre lexicographique, et `1` si `s2` est avant `s1` dans l'ordre lexicographique.

`value string_for_read : string -> string`

Renvoie une copie de l'argument, avec les caractères spéciaux représentés par des séquences d'échappement, en suivant les mêmes conventions que Caml Light.

9.17 `vect` : opérations sur les tableaux

`value vect_length : 'a vect -> int`

Renvoie la longueur (nombre d'éléments) du tableau donné.

`value vect_item : 'a vect -> int -> 'a`

`vect_item v n` renvoie l'élément numéro `n` du tableau `v`. Le premier élément est l'élément numéro 0. Le dernier élément a le numéro `vect_length v - 1`. Déclenche `Invalid_argument "vect_item"` si `n` est en dehors de l'intervalle 0, `(vect_length v - 1)`. On peut aussi écrire `v.(n)` au lieu de `vect_item v n`.

`value vect_assign : 'a vect -> int -> 'a -> unit`

`vect_assign v n x` modifie physiquement le tableau `v`, en remplaçant l'élément numéro `n` par `x`. Déclenche `Invalid_argument "vect_assign"` si `n` est en dehors de l'intervalle 0, `vect_length v - 1`. On peut aussi écrire `v.(n) <- x` au lieu de `vect_assign v n x`.

`value make_vect : int -> 'a -> 'a vect`

`make_vect n x` renvoie un nouveau tableau de longueur `n`, initialisé avec des éléments tous physiquement égaux à `x`, au sens de l'opérateur `==`.

`value make_matrix : int -> int -> 'a -> 'a vect vect`

`make_matrix dimx dimy e` renvoie un tableau à deux dimensions (un tableau de tableaux) avec pour première dimension `dimx` et pour seconde dimension `dimy`. Initialement, tous les éléments de ce tableau sont physiquement égaux à `x`, au sens de l'opérateur `==` du module `eq`, section 9.4. On accède à l'élément `(x,y)` de la matrice `m` grâce à la notation `m.(x).(y)`.

`value concat_vect : 'a vect -> 'a vect -> 'a vect`

`concat_vect v1 v2` renvoie un nouveau tableau contenant la concaténation des tableaux `v1` et `v2`.

`value sub_vect : 'a vect -> int -> int -> 'a vect`

`sub_vect v debut long` renvoie un nouveau tableau de longueur `long`, contenant les éléments numéro `debut` jusqu'à `debut + long - 1` du tableau `v`. Déclenche `Invalid_argument "sub_vect"` si `debut` et `long` ne désignent pas un sous-tableau valide de `v`, c'est-à-dire si `debut < 0`, ou `long < 0`, ou `debut + long > vect_length v`.

`value copy_vect : 'a vect -> 'a vect`

> `copy_vect v` renvoie une copie du tableau `v`, c'est-à-dire un nouveau tableau contenant les mêmes éléments que `v`.

`value fill_vect : 'a vect -> int -> int -> 'a -> unit`

> `fill_vect v deb long x` modifie physiquement le tableau `v`, en remplaçant les éléments numéro `deb` à `deb + long - 1` par `x`. Déclenche `Invalid_argument "fill_vect"` si `deb` et `long` ne désignent pas un sous-tableau valide de `v`.

`value blit_vect: 'a vect -> int -> 'a vect -> int -> int -> unit`

> `blit_vect v1 o1 v2 o2 long` copie `long` éléments du tableau `v1`, à partir de l'élément numéro `o1`, dans le tableau `v2`, à partir de l'élément numéro `o2`. Cette fonction marche correctement même si `v1` et `v2` désignent le même tableau, et les sous-tableaux source et destination se recouvrent. Déclenche `Invalid_argument "blit_vect"` si `o1` et `long` ne désignent pas un sous-tableau valide de `v1`, ou si `o2` et `long` ne désignent pas un sous-tableau valide de `v2`.

`value list_of_vect : 'a vect -> 'a list`

`list_of_vect v` renvoie la liste de tous les éléments de `v`, c'est-à-dire `[v.(0); v.(1); ...; v.(vect_length v - 1)]`.

`value vect_of_list : 'a list -> 'a vect`

`vect_of_list l` renvoie un nouveau tableau contenant les éléments de la liste `l`.

`value map_vect : ('a -> 'b) -> 'a vect -> 'b vect`

`map_vect f v` applique la fonction `f` à tous les éléments de `v` et construit un tableau contenant les résultats retournés par `f` :
`[| f v.(0); f v.(1); ...; f v.(vect_length v - 1) |]`.

`value map_vect_list : ('a -> 'b) -> 'a vect -> 'b list`

`map_vect_list f v` applique la fonction `f` à tous les éléments de `v` et construit une liste contenant les résultats retournés par `f` :
`[ f v.(0); f v.(1); ...; f v.(vect_length v - 1) ]`.

`value do_vect : ('a -> 'b) -> 'a vect -> unit`

`do_vect f v` applique tour à tour la fonction `f` à tous les éléments de `v`, ignorant tous les résultats :
`f v.(0); f v.(1); ...; f v.(vect_length v - 1); ()`.

10
La bibliothèque d'utilitaires

Ce chapitre décrit les modules de la bibliothèque d'utilitaires de Caml Light. Comme les modules de la bibliothèque de base, les modules de la bibliothèque d'utilitaires sont automatiquement liés avec les fichiers de code compilé de l'utilisateur par la commande `camlc`. Les fonctions définies par cette bibliothèque sont donc utilisables dans les programmes indépendants sans que l'on ait à ajouter aucun fichier `.zo` sur la ligne de commande de la phase d'édition de liens. De la même façon, en utilisation interactive ces globaux sont utilisables dans toutes les phrases entrées par l'utilisateur sans que l'on ait à charger aucun fichier de code au préalable.

Au contraire des modules de la bibliothèque de base, les modules de la bibliothèque d'utilitaires ne sont pas automatiquement "ouverts" au début de chaque compilation et au lancement du système interactif. Il est donc nécessaire d'utiliser des identificateurs qualifiés pour faire référence aux fonctions fournies par ces modules, ou d'ajouter des directives `#open`.

Conventions

Pour faciliter la recherche, les modules sont présentés par ordre alphabétique. Pour chaque module, les déclarations de son fichier d'interface sont imprimées une par une en police "machine à écrire", suivie chacune par un court commentaire. Tous les modules et les identificateurs qu'ils exportent sont indexés à la fin de cet ouvrage.

10.1 `arg` : analyse des arguments de la ligne de commande

Ce module fournit un mécanisme général pour extraire les options et arguments de la ligne de commande d'un programme.

Syntaxe de la ligne de commandes

Un mot-clé est une chaîne de caractères qui commence avec un `-`. Une option est un mot-clé seul ou suivi d'un argument. Il y a quatre genres de mots-clés : Unit, String, Int et Float. Les mots-clés du genre Unit ne prennent pas d'argument. Les mots-clés du genre String, Int, et Float prennent pour argument le mot suivant sur

la ligne de commande. Les arguments non précédés par un mot-clé sont appelés arguments anonymes. Exemples : (`foo` est supposé être le nom de la commande)

`foo -flag`	(une option sans argument)
`foo -int 1`	(une option entière avec argument `1`)
`foo -string foobar`	(une option chaîne avec argument `"foobar"`)
`foo -float 12.34`	(une option flottante avec argument `12.34`)
`foo 1 2 3`	(trois arguments anonymes : `"1"`, `"2"` et `"3"`)

`foo 1 -flag 3 -string bar`

(un argument anonyme, une option sans argument, un argument anonyme, une option chaîne avec argument `"bar"`)

```
type spec =
  String of (string -> unit)
| Int of (int -> unit)
| Unit of (unit -> unit)
| Float of (float -> unit)
```

Le type somme qui décrit le comportement associé à un mot-clé.

```
value parse : (string * spec) liste -> (string -> unit) -> unit
```

`parse speclist anonfun` analyse la ligne de commande et appelle les fonctions de `speclist` quand un mot-clé est détecté, et `anonfun` sur les arguments anonymes. Les fonctions sont appelées dans l'ordre d'apparition des mots sur la ligne de commande. Les chaînes de la liste `(string * spec) list` sont les mots-clés. Ils doivent commencer par un `-`, sinon ils sont ignorés. Pour que l'utilisateur puisse donner des arguments anonymes qui commencent par un `-`, mettre par exemple `("--", String anonfun)` dans la liste `speclist`.

```
exception Bad of string
```

Les fonctions de `speclist` ou `anonfun` peuvent déclencher `Bad` avec un message d'erreur pour rejeter des arguments erronés.

10.2 `filename` : opérations sur les noms de fichiers

```
value current_dir_name : string
```

Le nom conventionnel du répertoire courant (par exemple « `.` » en Unix).

```
value concat : string -> string -> string
```

`concat rép fich` renvoie un nom de fichier qui désigne le fichier `fich` dans le répertoire `rép`.

```
value is_absolute : string -> bool
```

Renvoie `true` si le nom de fichier est absolu ou commence par une référence explicite au répertoire courant (`./` ou `../` en Unix), et `false` sinon.

```
value check_suffix : string -> string -> bool
```

`check_suffix nom suff` renvoie `true` si le nom de fichier `nom` finit par le suffixe `suff`.

```
value chop_suffix : string -> string -> string
```

`chop_suffix nom suff` retire le suffixe `suff` du nom de fichier `nom`. Le comportement n'est pas spécifié si `nom` ne se termine pas par le suffixe `suff`.

```
value basename : string -> string
value dirname : string -> string
```

Coupe un nom de fichier en un nom de répertoire et un nom de base, respectivement. `concat (dirname nom) (basename nom)` renvoie un nom de fichier qui est équivalent à `nom`. De plus, après avoir fixé le répertoire courant à `dirname nom` (à l'aide de `sys__chdir`), les références à `basename nom` (qui est un nom de fichier relatif) désignent le même fichier que `nom` avant l'appel à `chdir`.

10.3 `genlex` : un analyseur lexical générique

Ce module fournit un analyseur lexical d'usage général, sous la forme d'une fonction transformant un flux de caractères en flux de lexèmes. Cet analyseur suit des conventions lexicales fixes pour ce qui est des identificateurs, des littéraux, des commentaires, etc., mais l'ensemble des mots-clés réservés est paramétrable. Les conventions lexicales utilisées sont proches de celles du langage Caml.

```
type token =
    Kwd of string
  | Ident of string
  | Int of int
  | Float of float
  | String of string
  | Char of char
```

Le type des lexèmes. Les classes lexicales sont : `Int` et `Float` pour les entiers littéraux et les décimaux littéraux, `String` pour les chaînes de caractères littérales entre guillemets (`"..."`), `Char` pour les caractères littéraux entre apostrophes inverses (`` `...` ``), `Ident` pour les identificateurs, `Kwd` pour les mots-clés. Les identificateurs sont de deux sortes : ou bien une suite de lettres, chiffres et caractères soulignés, ou bien une suite de caractères « opérateurs » comme `+` et `*`. Les mots-clés sont ou bien des identificateurs, ou bien des caractères « spéciaux » comme `(` et `}`.

```
value make_lexer: string list -> (char stream -> token stream)
```

Construit la fonction d'analyse lexicale, étant donné la liste des mots-clés. Lorsque l'analyseur lit un identificateur `id`, il renvoie `Kwd id` si `id` appartient à la liste des mots-clés, et `Ident id` sinon. Lorsque l'analyseur lit un caractère « spécial » `c` (ni lettre, ni chiffre, ni opérateur), il renvoie `Kwd c` si ce caractère appartient à la liste des mots-clés, et déclenche une erreur (exception `Parse_error`) sinon.

L'analyseur ignore les blancs (espaces, retours-chariot, etc.) et les commentaires délimités par `(*` et `*)`. Les blancs séparent les lexèmes adjacents. Les commentaires peuvent être emboîtés.

Exemple : un analyseur lexical adapté à une calculatrice « quatre opérations » s'obtient par

```
let lexer = make_lexer ["+";"-";"*";"/";"let";"="; "("; ")"];;
```

L'analyseur syntaxique associé se présente sous la forme d'une fonction du type `token stream` vers le type `int` (par exemple) et se présente typiquement sous la forme :

```
let parse_expr = function
    [< 'Int n >] -> n
  | [< 'Kwd "("; parse_expr n; 'Kwd ")" >] -> n
  | [< parse_expr n1; (parse_end n1) n2 >] -> n2
and parse_end n1 = function
    [< 'Kwd "+"; parse_expr n2 >] -> n1+n2
  | [< 'Kwd "-"; parse_expr n2 >] -> n1-n2
  | ...
```

10.4 `hashtbl` : tables de hachage

Les tables de hachage sont des tables d'association par hachage, avec modification physique.

`type ('a, 'b) t`

Le type des tables associant des éléments du type `'b` à des clés de type `'a`.

`value new : int -> ('a,'b) t`

`new n` crée une nouvelle table de hachage, initialement vide, de taille `n`. La table grossit si nécessaire, donc la valeur de `n` n'est qu'indicative. Prendre un nombre premier pour `n` rend le hachage plus efficace.

`value clear : ('a, 'b) t -> unit`

Vide une table de hachage.

`value add : ('a, 'b) t -> 'a -> 'b -> unit`

`add tbl x y` lie la valeur `y` à la clé `x` dans la table `tbl`. Les précédentes liaisons de `x` ne sont pas détruites, mais simplement cachées, si bien qu'après l'appel `remove tbl x`, la liaison précédente de `x`, si elle existe, est rétablie. (C'est la sémantique des listes d'association.)

`value find : ('a, 'b) t -> 'a -> 'b`

`find tbl x` renvoie la donnée associée à `x` dans `tbl`, ou déclenche `Not_found` si `x` n'est pas lié.

`value find_all : ('a, 'b) t -> 'a -> 'b list`

`find_all tbl x` renvoie la liste de toutes les données associées à `x` dans `tbl`. La liaison courante est retournée en premier, puis les liaisons précédentes, en ordre inverse d'introduction dans la table.

`value remove : ('a, 'b) t -> 'a -> unit`

`remove tbl x` retire la liaison courante de `x` dans `tbl`, rétablissant la liaison précédente si elle existe. Ne fait rien si `x` n'est pas lié dans `tbl`.

`value do_table : ('a -> 'b -> 'c) -> ('a, 'b) t -> unit`

`do_table f tbl` applique `f` à toutes les liaisons dans la table `tbl`, en ignorant tous les résultats. `f` reçoit la clé pour premier argument, et la valeur associée pour second argument. L'ordre dans lequel les liaisons sont passées à `f` n'est pas spécifié.

La primitive polymorphe de hachage

`value hash : 'a -> int`

`hash x` associe un entier positif à toute valeur de tout type. Il est garanti que si `x = y`, alors `hash x = hash y`. De plus, `hash` termine toujours, même sur des structures cycliques.

`value hash_param : int -> int -> 'a -> int`

`hash_param n m x` calcule une valeur de hachage pour `x`, avec les mêmes propriétés que pour `hash`. Les deux paramètres supplémentaires `n` et `m` donnent un contrôle plus précis sur le hachage. Le hachage effectue une traversée en profondeur d'abord, de droite à gauche, de la structure `x`. Le parcours s'arrête quand `n` nœuds significatifs, ou bien `m` nœuds significatifs ou non, ont été visités. Les nœuds significatifs sont : les entiers, les flottants, les chaînes de caractères, les caractères, les booléens et les constructeurs constants. Accroître `m` et `n` signifie accroître le nombre de nœuds pris en compte pour le calcul final de la valeur de hachage, et donc éventuellement diminuer le nombre de collisions. Cependant, le hachage sera plus lent. Les paramètres `m` et `n` règlent donc le compromis entre la précision et la vitesse.

10.5 `lexing` : la bibliothèque d'exécution des analyseurs lexicaux engendrés par camllex

Tampons d'analyseurs lexicaux

`type lexbuf`

Le type des tampons d'analyseurs lexicaux. Un tampon d'analyseur lexical est l'argument passé aux fonctions d'analyse lexicale définies par les analyseurs engendrés par `camllex`. Le tampon contient l'état courant de l'analyseur, plus une fonction pour remplir le tampon.

```
value create_lexer_channel : in_channel -> lexbuf
```

Crée un tampon sur le canal d'entrée donné. `create_lexer_channel can` renvoie un tampon qui lit depuis le canal d'entrée `can`, à la position courante de lecture.

```
value create_lexer_string : string -> lexbuf
```

Crée un tampon qui lit depuis la chaîne donnée. La lecture commence à partir du premier caractère de la chaîne. Une condition fin-de-fichier est produite quand la fin de la chaîne est atteinte.

```
value create_lexer : (string -> int -> int) -> lexbuf
```

Crée un tampon avec pour méthode de lecture la fonction à trois paramètres donnée. Quand l'analyseur a besoin de caractères, il appelle la fonction donnée, en lui fournissant une chaîne de caractères `s` et un compte de caractères `n`. La fonction doit placer `n` (ou moins de `n`) caractères dans `s`, en commençant au caractère numéro 0, et renvoyer le nombre de caractères fournis. Retourner la valeur 0 produit une condition fin-de-fichier.

Fonctions pour les actions sémantiques des analyseurs lexicaux

Les fonctions suivantes peuvent être appelées à partir des actions sémantiques des analyseurs (le code Caml Light entre accolades qui calcule la valeur retournée par les fonctions d'analyse). Elles donnent accès à la chaîne de caractères filtrée par l'expression rationnelle associée à l'action sémantique. Ces fonctions doivent être appliquées à l'argument `lexbuf` qui, dans le code engendré par `camllex`, est lié au tampon passé à la fonction d'analyse syntaxique.

```
value get_lexeme : lexbuf -> string
```

`get_lexeme lexbuf` renvoie la chaîne filtrée par l'expression rationnelle.

```
value get_lexeme_char : lexbuf -> int -> char
```

`get_lexeme_char lexbuf i` renvoie le caractère numéro `i` de la chaîne filtrée.

```
value get_lexeme_start : lexbuf -> int
```

`get_lexeme_start lexbuf` renvoie la position dans le tampon d'entrée du premier caractère de la chaîne filtrée. Le premier caractère lu a la position 0.

```
value get_lexeme_end : lexbuf -> int
```

`get_lexeme_start lexbuf` renvoie la position dans le tampon d'entrée du caractère suivant le dernier caractère de la chaîne filtrée. Le premier caractère lu a la position 0.

10.6 `parsing` : la bibliothèque d'exécution des analyseurs syntaxiques engendrés par camlyacc

```
value symbol_start : unit -> int
value symbol_end : unit -> int
```

Appelées depuis la partie « action » d'une règle de grammaire, ces fonctions renvoient la position de la chaîne qui filtre le membre gauche de la règle : `symbol_start()` renvoie la position du premier caractère ; `symbol_end()` renvoie la position du dernier caractère, plus un. Le premier caractère lu a la position 0.

```
value rhs_start: int -> int
value rhs_end: int -> int
```

Ces fonctions sont analogues à `symbol_start` et `symbol_end`, mais renvoient la position de la chaîne qui filtre la $n^{\text{ième}}$ composante du membre droit de la règle, où n est l'entier argument de `rhs_start` et `rhs_end`. La première composante correspond à $n = 1$.

```
value clear_parser : unit -> unit
```

Vide la pile de l'analyseur syntaxique. Appeler cette fonction au retour d'une fonction d'analyse syntaxique supprime tous les pointeurs contenus dans la pile de l'analyseur, liés à des objets construits par les actions sémantiques lors de l'analyse et désormais inutiles. Ce n'est pas indispensable, mais réduit les besoins en mémoire des programmes.

10.7 `printexc` : affichage d'exceptions imprévues

```
value f: ('a -> 'b) -> 'a -> 'b
```

`f fn x` applique `fn` à `x` et renvoie le résultat obtenu. Si l'évaluation de `fn x` déclenche une exception, le nom de l'exception est imprimé sur la sortie d'erreur standard, et le programme s'arrête avec le code de sortie 2. L'usage habituel est `printexc__f main ()`, où `main` (de type `unit -> unit`) est le point d'entrée d'un programme indépendant. Ainsi l'on peut rattraper et imprimer toutes les exceptions inattendues. Pour que les exceptions soient correctement imprimées, le programme doit avoir été lié avec l'option `-g` du compilateur indépendant.

10.8 `printf` : impression formatée

```
value fprintf: out_channel -> string -> 'a
```

`fprintf sortie format arg1 ... argN` met en forme les arguments `arg1` à `argN` en suivant le format de la chaîne `format`, et sort la chaîne résultat sur le canal `sortie`. Le format est une chaîne de caractères qui contient deux types d'objets : des caractères, qui sont simplement copiés sur le canal de sortie, et des spécifications de conversion, qui chacune convertissent et impriment un argument. Les spécifications de conversion consistent en un caractère `%`, suivi par des options de mise en forme (largeurs de champs), suivies par un caractère de conversion. Voici les différents caractères de conversion et leur signification :

`d` ou `i` : convertit un entier en un décimal signé.

`u` : convertit un entier en un décimal non signé.

`x` : convertit un entier en un hexadécimal non signé, en lettres minuscules.

`X` : convertit un entier en un hexadécimal non signé, en lettres majuscules.

`s` : insère une chaîne.

`c` : insère un caractère.

`f` : convertit un flottant en notation décimale, dans le style `dddd.ddd`.

`e` ou `E` : convertit un flottant en notation décimale, dans le style `d.ddd e+-dd` (mantisse et exposant).

`g` ou `G` : convertit un flottant en notation décimale, dans le style `f` ou `e` suivant ce qui est le plus compact.

`b` : convertit un booléen en la chaîne `true` ou `false`.

On se reportera à la fonction `printf` de la bibliothèque C pour la signification des options de mise en forme. L'exception `Invalid_argument` est déclenchée si les types des arguments donnés ne correspondent pas au format, ou si trop d'arguments sont fournis. En revanche, si pas assez d'arguments sont fournis, l'impression s'arrête juste avant de convertir le premier argument manquant. Cette fonction n'est pas sûre, puisqu'elle renvoie un objet de type `'a`, qui peut être mal utilisé ensuite. Il est recommandé de toujours utiliser `fprintf` à l'intérieur d'une séquence, comme dans `fprintf "%d" n; ()`.

`value printf: string -> 'a`

Analogue à `fprintf`, mais sort sur `std_out`.

`value fprint: out_channel -> string -> unit`

Imprime la chaîne donnée sur le canal de sortie donné, sans aucun formatage. C'est la même fonction que `output_string` du module `io`.

`value print: string -> unit`

Imprime la chaîne donnée sur `std_out`, sans aucun formatage. C'est la même fonction que `print_string` du module `io`.

10.9 queue : files d'attente

Ce module implémente les files d'attente, qui suivent la stratégie premier entré, premier sorti, avec modification en place. Ces files sont aussi appelées FIFO pour *first-in first-out*.

`type 'a t`

Le type des files contenant des éléments de type `'a`.

`exception Empty`

Déclenchée quand `take` est appliquée à une file vide.

`value new: unit -> 'a t`

Renvoie une nouvelle file, initialement vide.

`value add: 'a -> 'a t -> unit`

`add x q` ajoute l'élément `x` à la fin de la file `q`.

`value take: 'a t -> 'a`

`take q` supprime et renvoie le premier élément de la file `q`, ou déclenche `Empty` si la file est vide.

`value peek: 'a t -> 'a`

`peek q` renvoie le premier élément de la file `q`, sans le supprimer de la file, ou déclenche `Empty` si la file est vide.

`value clear : 'a t -> unit`

Supprime tous les éléments d'une file.

`value length: 'a t -> int`

Renvoie le nombre d'éléments d'une file.

`value iter: ('a -> 'b) -> 'a t -> unit`

`iter f q` applique `f` à tous les éléments de la file `q` tour à tour, en commençant par l'élément le plus anciennement entré, pour terminer par l'élément le plus récemment entré. La file elle-même n'est pas modifiée.

10.10 `random` : générateur de nombres pseudo-aléatoires

`value init : int -> unit`

Initialise le générateur aléatoire, en utilisant pour germe (en anglais *seed*) l'argument fourni. Le même germe produit toujours la même suite de nombres.

`value int : int -> int`

`random__int limite` renvoie un nombre entier aléatoire entre 0 (compris) et `limite` (non compris). L'entier `limite` doit être inférieur à 2^{30}.

`value float : float -> float`

`random__float limite` renvoie un nombre flottant aléatoire entre 0 (compris) et `limite` (non compris).

10.11 `sort` : tri et fusion de listes

`value sort : ('a -> 'a -> bool) -> 'a list -> 'a list`

Trie une liste en ordre croissant selon un ordre donné. Le prédicat doit renvoyer `true` si son premier argument est inférieur ou égal à son second argument.

`value merge: ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list`

Fusionne deux listes selon le prédicat donné. En supposant que les deux listes arguments sont triées d'après le prédicat, `merge` renvoie une liste triée contenant les éléments des deux listes. Le comportement n'est pas spécifié si les arguments ne sont pas triés.

10.12 `stack` : piles

Ce module implémente les piles qui suivent la stratégie dernier entré, premier sorti, avec modification en place. Ces piles sont aussi appelées LIFO pour *last-in first-out*.

`type 'a t`

Le type des piles contenant des éléments de type `'a`.

`exception Empty`

Déclenchée quand `pop` est appliquée à une pile vide.

`value new: unit -> 'a t`

Renvoie une nouvelle pile, initialement vide.

`value push: 'a -> 'a t -> unit`

`push x s` ajoute l'élément `x` au sommet de la pile `s`.

`value pop: 'a t -> 'a`

`pop s` supprime et renvoie l'élément au sommet de la pile `s`, ou déclenche `Empty` si la pile est vide.

`value clear : 'a t -> unit`

Supprime tous les éléments d'une pile.

`value length: 'a t -> int`

Renvoie le nombre d'éléments d'une pile.

`value iter: ('a -> 'b) -> 'a t -> unit`

`iter f s` applique `f` à tous les éléments de la pile `s` tour à tour, en commençant par l'élément au sommet de la pile, pour terminer par l'élément au fond de la pile. La pile elle-même n'est pas modifiée.

10.13 `sys` : interface système

Ce module fournit une interface simplifiée avec le système d'exploitation.

`exception Sys_error of string`

Déclenchée par certaines fonctions des modules `sys` et `io`, quand l'appel système sous-jacent échoue. L'argument de `Sys_error` est une chaîne qui décrit l'erreur. Les textes des messages d'erreur dépendent de l'implémentation, et l'on ne peut s'y fier pour rattraper une erreur système particulière.

```
value command_line : string vect
```

Les arguments passés au processus sur la ligne de commande. Le premier élément est le nom de commande sous lequel le programme a été appelé.

```
value interactive : bool
```

La variable booléenne `interactive` est vraie si le code s'exécute sous le système interactif, et fausse s'il s'exécute comme un programme indépendant.

```
type file_perm == int
value s_irusr : file_perm
value s_iwusr : file_perm
value s_ixusr : file_perm
value s_irgrp : file_perm
value s_iwgrp : file_perm
value s_ixgrp : file_perm
value s_iroth : file_perm
value s_iwoth : file_perm
value s_ixoth : file_perm
value s_isuid : file_perm
value s_isgid : file_perm
value s_irall : file_perm
value s_iwall : file_perm
value s_ixall : file_perm
```

Droits d'accès aux fichiers. `r` désigne le droit en lecture, `w` le droit en écriture, `x` le droit d'exécution. `usr` indique les droits de l'utilisateur propriétaire du fichier, `grp` les droits du groupe propriétaire du fichier, `oth` les droits des autres. `isuid` et `isgid` sont pour les fichiers *set-user-id* et *set-group-id*, respectivement.

```
type open_flag =
    O_RDONLY      (* ouverture en lecture seule *)
  | O_WRONLY      (* ouverture en écriture seule *)
  | O_RDWR        (* ouverture en lecture et écriture *)
  | O_APPEND      (* ouverture en ajout *)
  | O_CREAT       (* création du fichier s'il n'existe pas *)
  | O_TRUNC       (* troncature du fichier à 0 s'il existe *)
  | O_EXCL        (* échec si le fichier existe *)
  | O_BINARY      (* ouverture en mode binaire *)
  | O_TEXT        (* ouverture en mode texte *)
```

Les commandes pour la fonction `open`.

```
value exit : int -> 'a
```

Achève le programme et renvoie le code de retour (en anglais *status code*) donné au système d'exploitation. Au contraire de la fonction `exit` du module `io`, cette fonction ne vide pas les canaux de sortie et d'erreur standard.

`value open : string -> open_flag list -> file_perm -> int`

Ouvre un fichier. Le second argument est le mode d'ouverture. Le troisième argument précise les droits à utiliser si le fichier doit être créé. Le résultat est un descripteur de fichier, ouvert sur le fichier.

`value close : int -> unit`

Ferme un descripteur de fichier.

`value remove : string -> unit`

Supprime le fichier de nom donné du système de fichiers.

`value rename : string -> string -> unit`

Renomme un fichier. Le premier argument est l'ancien nom et le second est le nouveau.

`value getenv : string -> string`

Renvoie la valeur associée à une variable d'environnement dans l'environnement du processus. Déclenche `Not_found` si la variable n'est pas liée.

`value chdir : string -> unit`

Change le répertoire de travail courant du processus.

`exception Break`

Déclenchée lors d'une interruption de l'utilisateur si `catch_break` est activée.

`value catch_break : bool -> unit`

`catch_break` décide si les interruptions de l'utilisateur mettent fin au programme (comportement par défaut) ou déclenchent simplement l'exception `Break`. Après `catch_break true`, l'exception `Break` est déclenchée à chaque interruption utilisateur. Après `catch_break false`, le système d'exploitation met fin au programme en cas d'interruption de l'utilisateur.

11
La bibliothèque graphique

Ce chapitre décrit les primitives graphiques portables fournies en standard dans les implémentations de Caml Light sur micro-ordinateurs.

Unix : Sur les stations de travail Unix munies du système X Windows, ces primitives graphiques sont fournies par une bibliothèque qui se trouve dans le répertoire `contrib/libgraph` de la distribution Caml Light. Le tracé a lieu dans une fenêtre X séparée qui est créée quand la fonction `open_graph` est appelée. La commande `camlgraph` fournie par la bibliothèque intègre les primitives graphiques dans une boucle interactive. Les fonctionnalités de la bibliothèque graphique sont alors les mêmes que celles des autres versions de Caml Light. Les fichiers de la distribution expliquent comment installer la bibliothèque graphique et comment la lier avec vos propres programmes pour créer des programmes indépendants.

Mac : Les primitives graphiques sont disponibles à partir de l'application indépendante qui exécute le système interactif. Elles ne sont pas disponibles à partir de programmes compilés par `camlc` et qui tournent sous l'interpréteur de commandes MPW. Les tracés ont lieu dans une fenêtre indépendante, qu'on peut rendre visible à l'aide de l'option "Show graphic window" du menu.

PC : Les primitives graphiques sont disponibles en interactif et dans les programmes compilés séparément. La primitive `open_graph` passe tout l'écran en mode graphique. Sous le système interactif, les phrases de l'utilisateur et les réponses du système sont également imprimées sur l'écran graphique, en effaçant éventuellement certaines parties du dessin et en faisant défiler l'écran quand l'impression atteint le bas de l'écran.

Les coordonnées de l'écran sont interprétées comme dans la figure 11.1. Le système de coordonnées utilisé est le même qu'en mathématiques : y augmente de bas en haut et les angles sont mesurés en degrés dans le sens trigonométrique. Conceptuellement le tracé a lieu dans un plan infini, dont l'écran montre une partie ; les tracés en dehors de l'écran sont donc permis.

Voici les spécifications des modes graphiques autorisés par `open_graph` sur les différentes implémentations de cette bibliothèque.

Unix : L'argument d'`open_graph` est une chaîne contenant le nom de l'écran (en anglais *display*) du serveur X Windows auquel se connecter, puis un espace,

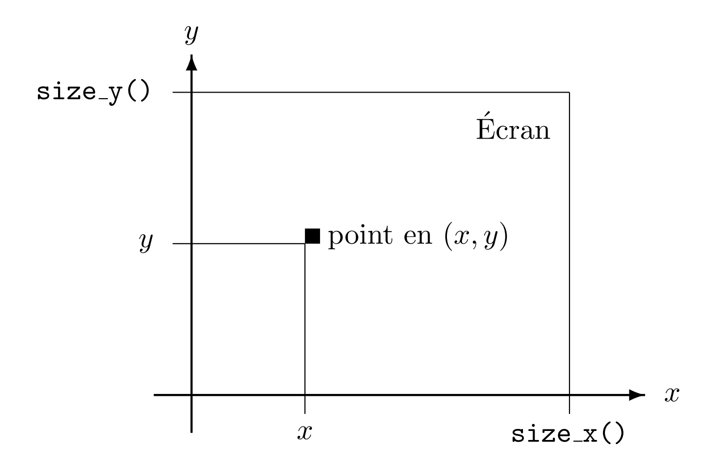


Figure 11.1: Le repère de l'écran graphique

puis une spécification de géométrie X Windows standard. Le nom d'écran et/ou la géométrie peuvent être omis. Exemples :

`open_graph "foo :0"`
: se connecte à l'écran `foo:0` et crée une fenêtre avec la géométrie par défaut.

`open_graph "foo :0 300x100+50-0"`
: se connecte à l'écran `foo:0` et crée une fenêtre de 300 pixels de large sur 100 pixels de haut, en $(50, 0)$.

`open_graph " 300x100+50-0"`
: se connecte à l'écran par défaut et crée une fenêtre de 300 pixels de large sur 100 pixels de haut, en $(50, 0)$.

`open_graph ""`
: se connecte à l'écran par défaut et crée une fenêtre avec la géométrie par défaut.

Mac : L'argument d'`open_graph` est ignoré.

86 : La version PC 86 est compatible avec les cartes graphiques CGA, EGA et VGA. L'argument d'`open_graph` est interprété comme suit :

Argument	Mode graphique
`"cga"`	320×200, 4 couleurs
`"ega_low"`	640×200, 16 couleurs
`"ega_high"`	640×350, 16 couleurs
`"vga_low"`	640×200, 16 couleurs
`"vga_med"`	640×350, 16 couleurs
`"vga_high"`	640×480, 16 couleurs
autre chose	la résolution la plus fine possible

386 : La version PC 386 fonctionne avec les cartes graphiques VGA et la plupart des cartes SuperVGA, pourvu que le pilote de graphique adéquat ait été installé. (Voir les instructions d'installation au chapitre 12.) Seuls les modes 256 couleurs sont utilisables. Cela implique une résolution de 320×200 sur VGA standard

et de 640×480 à 1024×768 sur SuperVGA. L'argument d'`open_graph` est interprété comme suit :

Argument	Mode graphique
`"highest"`	la résolution la plus fine possible
`"noninterlaced"`	la résolution la plus fine possible, sans entrelacement
`"320x200"`	mode 320×200 VGA standard
"wxh"	au moins w par h points, si possible
autre chose	la résolution par défaut, fixée par la variable `GO32` (voir les instructions d'installation)

11.1 `graphics` : primitives graphiques portables

`exception Graphic_failure of string`

Déclenchée par les fonctions de ce module quand elles rencontrent une erreur.

Initialisations

`value open_graph: string -> unit`

Montre la fenêtre graphique, ou passe l'écran en mode graphique. La fenêtre graphique est effacée. La chaîne argument sert à donner des informations optionnelles sur le mode graphique désiré, la taille de la fenêtre graphique, etc. Son interprétation dépend de l'implémentation. Si la chaîne fournie est vide, un mode par défaut raisonnable est utilisé.

`value close_graph: unit -> unit`

Supprime la fenêtre graphique ou remet l'écran en mode texte.

`value clear_graph : unit -> unit`

Efface la fenêtre graphique.

`value size_x : unit -> int`
`value size_y : unit -> int`

Renvoie la taille de la fenêtre graphique. Les coordonnées des points de l'écran vont de `0 à size_x()-1` en abscisses et de `0 à size_y()-1` en ordonnées.

Couleurs

`type color == int`

Une couleur est spécifiée par ses composantes R, V, B (R pour le rouge, V pour le vert et B pour le bleu). Chaque composante est un entier dans l'intervalle `0..255`. Les trois composantes sont groupées en un `int` : `0xRRVVBB`, où `RR` sont les deux chiffres hexadécimaux pour la composante rouge, `VV` pour la composante vert et `BB` pour la composante bleu.

`value rgb: int -> int -> int -> int`

`rgb r v b` renvoie l'entier qui code la couleur avec pour composante rouge `r`, pour composante verte `v` et pour composante bleue `b`. Les entiers `r`, `v` et `b` sont dans l'intervalle `0..255`.

`value set_color : color -> unit`

Fixe la couleur courante du tracé.

```
value black : color
value white : color
value red : color
value green : color
value blue : color
value yellow : color
value cyan : color
value magenta : color
```

Quelques couleurs prédéfinies.

`value background: color`

La couleur du fond de l'écran graphique (généralement soit noir, soit blanc). `clear_graph` peint l'écran avec cette couleur.

`value foreground: color`

La couleur de tracé initiale (généralement, noir si le fond de l'écran est blanc, et blanc si le fond est noir).

Tracés de points et de lignes

`value plot : int -> int -> unit`

Trace le point donné avec la couleur courante du tracé.

`value point_color : int -> int -> color`

Renvoie la couleur du point donné. Résultat non spécifié si le point est en dehors de l'écran.

`value moveto : int -> int -> unit`

Positionne le point courant.

`value current_point : unit -> int * int`

Renvoie la position du point courant.

`value lineto : int -> int -> unit`

`lineto x y` trace une ligne entre le point courant et le point `(x,y)` et déplace le point courant au point `(x,y)`.

`value draw_arc : int -> int -> int -> int -> int -> int -> unit`

`draw_arc x y rx ry a1 a2` trace un arc d'ellipse avec pour centre `(x,y)`, rayon horizontal `rx`, rayon vertical `ry`, de l'angle `a1` à l'angle `a2` (en degrés). Le point courant n'est pas modifié.

`value draw_ellipse : int -> int -> int -> int -> unit`

`draw_ellipse x y rx ry` trace une ellipse avec pour centre `x,y`, rayon horizontal `rx` et rayon vertical `ry`. Le point courant n'est pas modifié.

`value draw_circle : int -> int -> int -> unit`

`draw_circle x y r` trace un cercle avec pour centre `x,y` et rayon `r`. Le point courant n'est pas modifié.

`value set_line_width : int -> unit`

Fixe la largeur des points et des lignes tracés avec les fonctions ci-dessus.

Tracé de texte

`value draw_char : char -> unit`
`value draw_string : string -> unit`

Trace un caractère ou une chaîne de caractères avec le coin inférieur gauche à la position courante. Après le tracé, la position courante est fixée au coin inférieur droit du texte tracé.

`value set_font : string -> unit`
`value set_text_size : int -> unit`

Fixe la police et la taille des caractères utilisés pour tracer du texte. L'interprétation des arguments de `set_font` et `set_text_size` dépend de l'implémentation.

`value text_size : string -> int * int`

Renvoie les dimensions qu'aurait le texte donné s'il était tracé avec les polices et tailles courantes.

Remplissage

`value fill_rect : int -> int -> int -> int -> unit`

`fill_rect x y w h` remplit, avec la couleur courante, le rectangle dont le coin inférieur gauche est en `(x,y)` et dont la largeur et la hauteur sont respectivement `w` et `h`.

`value fill_poly : (int * int) vect -> unit`

Remplit le polygone donné avec la couleur courante. Le tableau contient les coordonnées des sommets du polygone.

`value fill_arc : int -> int -> int -> int -> int -> int -> unit`

Remplit un secteur d'ellipse avec la couleur courante. Les paramètres sont les mêmes que pour `draw_arc`.

`value fill_ellipse : int -> int -> int -> int -> unit`

Remplit une ellipse avec la couleur courante. Les paramètres sont les mêmes que pour `draw_ellipse`.

`value fill_circle : int -> int -> int -> unit`

Remplit un cercle avec la couleur courante. Les paramètres sont les mêmes que pour `draw_circle`.

Images

`type image`

Le type abstrait des images, en représentation interne. De façon externe, les images sont représentées par des matrices de couleurs (type `color vect vect`).

`value transp : color`

Dans les matrices de couleurs, cette couleur représente un point « transparent » : lorsqu'on trace une image, tous les points de l'écran correspondant à un point transparent dans l'image ne seront pas modifiés, alors que les autres points seront mis à la couleur du point correspondant dans l'image. Ceci permet de surimposer une image sur un fond existant.

`value make_image : color vect vect -> image`

Convertit la matrice de couleurs donnée en une image. Chaque sous-tableau représente une ligne horizontale. Tous les sous-tableaux doivent avoir la même longueur ; sinon, l'exception `Graphic_failure` est déclenchée.

`value dump_image : image -> color vect vect`

Convertit une image en une matrice de couleurs.

`value draw_image : image -> int -> int -> unit`

Trace l'image donnée en positionnant son coin inférieur droit au point donné.

`value get_image : int -> int -> int -> int -> image`

Capture le contenu d'un rectangle de l'écran en une image. Les paramètres sont les mêmes que pour `fill_rect`.

`value create_image : int -> int -> image`

`create_image l h` renvoie une nouvelle image de `l` points de large et de `h` points de haut, prête à être utilisée par `blit_image`. Le contenu initial de l'image est quelconque.

`value blit_image : image -> int -> int -> unit`

`blit_image img x y` copie des points de l'écran dans l'image `img`, en modifiant physiquement `img`. Les points copiés sont ceux à l'intérieur du rectangle de coin inférieur droit `x,y`, et de largeur et hauteur égaux à ceux de l'image.

Souris et événements clavier

```
type status =
  { mouse_x : int;    (* Coordonnée X de la souris *)
    mouse_y : int;    (* Coordonnée Y de la souris *)
    button : bool;    (* Vrai si un bouton est enfoncé *)
    keypressed : bool;(* Vrai si une touche a été enfoncée *)
    key : char }      (* Le caractère de la touche enfoncée *)
```

Pour rendre compte des événements.

```
type event =
    Button_down       (* Un bouton de la souris est enfoncé *)
  | Button_up         (* Un bouton de la souris est relâché *)
  | Key_pressed       (* Une touche est enfoncée *)
  | Mouse_motion      (* La souris est déplacée *)
  | Poll              (* Ne pas attendre; retourner aussitôt *)
```

Pour spécifier les événements attendus.

`value wait_next_event : event list -> status`

Attend l'avènement de l'un des événements spécifiés dans la liste d'événements donnée, et renvoie le statut de la souris et du clavier à ce moment. Si `Poll` est donné dans la liste d'événements, revient immédiatement avec le statut courant. (`Poll` signifie « sonder », d'où l'idée de « donner un simple coup de sonde » si cet événement est attendu.) Si le curseur de la souris est hors de la fenêtre graphique, les champs `mouse_x` et `mouse_y` de l'événement sont en dehors de l'intervalle `0..size_x()-1, 0..size_y()-1`. Les caractères tapés sont mémorisés dans un tampon et restitués un par un quand l'événement `Key_pressed` est spécifié.

Sonder la souris et le clavier

`value mouse_pos : unit -> int * int`

Renvoie la position du curseur de la souris relative à la fenêtre graphique. Si le curseur de la souris est hors de la fenêtre graphique, `mouse_pos()` renvoie un point en dehors de l'intervalle `0..size_x()-1, 0..size_y()-1`.

`value button_down : unit -> bool`

Renvoie `true` si le bouton de la souris est enfoncé, `false` sinon.

`value read_key : unit -> char`

Attend qu'une touche soit enfoncée, et renvoie le caractère correspondant. Les caractères tapés en avance sont conservés dans un tampon.

`value key_pressed : unit -> bool`

Renvoie `true` si un caractère est disponible dans le tampon d'entrée de `read_key`, et `false` sinon. Si `key_pressed` renvoie `true`, le prochain appel à `read_key` renverra un caractère sans attente.

Son

`value sound : int -> int -> unit`

`sound freq dur` émet un son de fréquence `freq` (en hertz) pendant une durée `dur` (en millisecondes). Sur le Macintosh, pour d'obscures raisons techniques, la fréquence est arrondie à la note la plus proche de la gamme tempérée.

IV

Annexes

12
Instructions d'installation

Ce chapitre vous explique comment installer Caml Light sur votre machine.

12.1 La version Unix

Matériel nécessaire. Toute machine fonctionnant sous le système d'exploitation Unix, avec un espace d'adressage non segmenté sur 32 ou 64 bits. 4M de RAM, 2M d'espace disque libre. La bibliothèque graphique nécessite le système de fenêtrage X11 version 4 ou 5.

Installation. La version Unix est distribuée sous forme de code source en un fichier compressé au format `tar` nommé `cl6unix.tar.Z`. Pour l'extraire, allez dans le répertoire où vous voulez mettre les fichiers source, transférez le fichier `cl6unix.tar.Z` et exécutez la commande

```
zcat cl6unix.tar.Z | tar xBf -
```

qui extrait les fichiers dans le répertoire courant. Le fichier `INSTALL` explique en détail comment configurer, compiler et installer Caml Light. Lisez-le et suivez les instructions.

Problèmes. Le fichier `INSTALL` explique les problèmes d'installation couramment rencontrés.

12.2 La version Macintosh

Matériel nécessaire. Tout Macintosh disposant d'au moins 1M de mémoire vive (2M sont recommandés) et tournant sous les versions 6 ou 7 du système. Environ 850K d'espace libre sur le disque dur. Le compilateur indépendant nécessite actuellement l'environnement de programmation Macintosh Programmer's Workshop (MPW) version 3.2. MPW est distribué par APDA, Apple's Programmers and Developers Association. Le fichier `READ ME` de la distribution contient les coordonnées de l'APDA.

Installation. Créez le dossier où vous voulez mettre les fichiers Caml Light. Double-cliquez sur le fichier `cl6macbin.sea` de la disquette de distribution. Une boîte de dialogue s'ouvre alors. Ouvrez le dossier où vous voulez mettre les fichiers Caml Light et cliquez sur le bouton `Extract`. Les fichiers de la distribution seront extraits de la disquette et copiés dans le dossier Caml Light.

Pour tester l'installation, double-cliquez sur l'application `Caml Light`. La fenêtre « Caml Light output » devrait alors afficher la bannière :

```
>       Caml Light version 0.6

#
```

Dans la fenêtre « Caml Light input », tapez « `1+2;;` » et appuyez sur la touche « Retour à la ligne ». La fenêtre « Caml Light output » devrait alors afficher

```
>       Caml Light version 0.6

#1+2;;
- : int = 3
#
```

Sélectionnez « Quit » dans le menu « File » pour quitter l'application et revenir au Finder.

Si vous disposez de MPW, vous pouvez installer les outils de compilation indépendante de la façon suivante : les outils et scripts du dossier `tools` doivent résider dans un endroit tel que MPW puisse les trouver et les reconnaître comme des commandes. Il y a deux façons d'obtenir ce résultat : vous pouvez soit copier les fichiers du dossier `tools` dans le dossier `Tools` ou `Scripts` de votre dossier MPW, soit laisser les fichiers dans le dossier `tools` et ajouter la ligne suivante au fichier `UserStartup` (en supposant que Caml Light réside dans le dossier `Caml Light` du disque dur `My HD`) :

```
Set Commands "{Commands},My HD:Caml Light:tools:"
```

Dans les deux cas, vous devez maintenant éditer le script `camlc` et remplacer la chaîne de caractères

```
Macintosh HD:Caml Light:lib:
```

(sur la première ligne) par le chemin d'accès réel au dossier `lib`. Par exemple si Caml Light réside dans le dossier `Caml Light` du disque dur `My HD`, la première ligne de `camlc` doit être :

```
Set stdlib "My HD:Caml Light:lib:"
```

Problèmes. Voici le problème le plus couramment rencontré :

`Cannot find file stream.zi` (Fichier `stream.zi` introuvable)
Affiché dans la fenêtre « Caml Light output », avec une boîte d'alerte qui vous signale que le programme Caml Light s'est terminé de façon anormale. C'est le symptôme d'une mauvaise installation. Le dossier nommé `lib` dans la distribution doit toujours être dans le même dossier que l'application `Caml Light`. On peut bien sûr déplacer l'application, mais pensez à déplacer le dossier `lib` en même temps. Sélectionnez « Quit » dans le menu « File » pour revenir au Finder.

12.3 Les versions PC

Deux versions sont distribuées pour le PC. La première, nommée ici « PC 8086 », tourne sur tous les PC, quel que soit leur processeur (8088, 8086, 80286, 80386, 80486, Pentium). La deuxième, nommée « PC 80386 », tourne en mode 32 bits protégé, est donc plus rapide et peut utiliser plus de mémoire que les 640K standard, mais ne tourne que sur les PC à processeurs 80386, 80486 ou Pentium. La disquette de distribution pour le PC contient les binaires pour les deux versions, dans deux fichiers d'archive au format ZIP, `cl6pc86.zip` et `cl6pc386.zip`.

Dans la suite, nous supposerons que la disquette de distribution est dans l'unité `A:` et que le disque dur sur lequel vous installez Caml Light est l'unité `C:`. Si ce n'est pas le cas, remplacez `A:` et `C:` par les noms d'unités appropriés.

12.3.1 La version PC 8086

Matériel nécessaire. Un PC tournant sous MS-DOS version 3.1 ou postérieure. 640K de RAM. Environ 800K de mémoire libre sur le disque dur. Les primitives graphiques nécessitent une carte vidéo CGA, EGA ou VGA.

Installation. Créez un répertoire sur le disque dur où vous mettrez le système Caml Light. Dans la suite nous supposerons que ce répertoire est `C:\CAML86`. Si vous choisissez un répertoire différent, remplacez `C:\CAML86` par le chemin absolu qui convient. Puis exécutez les commandes suivantes :

```
CD C:\CAML86
A:PKUNZIP -d A:CL5PC86
A:PKUNZIP -d A:CL5EXPLE
```

Prenez soin de bien choisir l'option `-d` de `pkunzip`. Cette commande extrait tous les fichiers du répertoire `C:\CAML86`. Ensuite, éditez le fichier de démarrage `C:\AUTOEXEC.BAT` pour :

- ajouter `C:\CAML86\BIN` à la variable `PATH`, c'est-à-dire transformer la ligne

  ```
  SET PATH=C:\DOS;...
  ```

  en

  ```
  SET PATH=C:\DOS;...;C:\CAML86\BIN
  ```

- insérer la ligne suivante :

  ```
  SET CAMLLIB=C:\CAML86\LIB
  ```

Puis sauvegardez le fichier `autoexec.bat` et redémarrez la machine. Pour tester l'installation, exécutez :

```
camlc -v
```

La commande `camlc` devrait imprimer quelque chose du genre :

```
The Caml Light system for the 8086 PC, version 0.6
  (standard library from C:\CAML86\LIB)
The Caml Light runtime system, version 0.6
The Caml Light compiler, version 0.6
The Caml Light linker, version 0.6
```

Exécutez ensuite

```
caml
```

La commande `caml` devrait afficher :

```
>       Caml Light version 0.6

#
```

En réponse au signe d'invite `#`, tapez :

```
quit();;
```

Cela devrait vous renvoyer à l'interpréteur de commandes DOS.

Problèmes. Voici quelques problèmes fréquemment rencontrés :

`Cannot find the bytecode file` (Fichier de code introuvable.)
`camlrun.exe : No such file` (Le fichier `camlrun.exe` n'existe pas.)

L'installation n'a pas été faite correctement. Vérifiez le fichier `AUTOEXEC.BAT` et la définition des variables `PATH` et `CAMLLIB`. N'oubliez pas qu'il faut redémarrer la machine pour que les modifications apportées à `AUTOEXEC.BAT` prennent effet.

`Out of memory` (La mémoire est pleine.)

Caml Light tient tout juste dans l'espace mémoire ridiculement petit fourni par MS-DOS. Vous devez donc vous assurer qu'un maximum des fatidiques 640K sont libres avant de lancer Caml Light. 500K de mémoire libre constituent le minimum absolu ; 600K sont encore à peine suffisants.

Pour libérer de la mémoire, supprimez le plus possible de pilotes de périphériques (en anglais *device drivers*) et de programmes résidents (*TSR*, pour *terminate and stay resident*). Pour ce faire, supprimez dans le fichier `CONFIG.SYS` les lignes qui chargent les pilotes de périphériques, et dans le fichier `AUTOEXEC.BAT` les lignes qui chargent les programmes résidents. Redémarrez ensuite la machine.

Une autre solution consiste à essayer de charger vos pilotes et TSR en dehors de la zone mémoire standard de 640K, en utilisant des gestionnaires de mémoire tels que QEMM ou EMM386 en MS-DOS 5. Pour plus de détails, consultez l'abondante littérature sur MS-DOS.

12.3.2 La version PC 80386

Matériel nécessaire. Un PC avec processeur 80386 ou 80486, tournant sous MS-DOS version 3.3 ou postérieure. 2M de RAM. Environ 1,2M de mémoire libre sur le disque dur. Les primitives graphiques nécessitent une carte vidéo VGA ou SuperVGA.

Installation. Créez un répertoire sur le disque dur où vous mettrez le système Caml Light. Dans la suite nous supposerons que ce répertoire est `C:\CAML386`. Si vous choisissez un répertoire différent, remplacez `C:\CAML386` par le chemin absolu approprié. Puis exécutez les commandes suivantes :

```
CD C:\CAML386
A:PKUNZIP -d A:CL5PC386
A:PKUNZIP -d A:CL5EXPLE
```

Prenez soin de bien choisir l'option `-d` de `pkunzip`. Cette commande extrait tous les fichiers du répertoire `C:\CAML386`.

Sélectionnez ou créez un répertoire où Caml Light pourra mettre ses fichiers temporaires. Beaucoup de machines ont déjà un répertoire `C:\TMP` pour cet usage. S'il n'existe pas, créez-le.

Pour la suite de la procédure de configuration vous devez déterminer deux choses :

- Votre machine possède-t-elle une arithmétique flottante câblée — c'est-à-dire un coprocesseur 387, un processeur 486DX, ou un coprocesseur 487SX ? (Si vous ne savez pas, faites comme si vous n'aviez pas d'arithmétique flottante câblée.)
- Quel type de carte SuperVGA avez-vous ? Les primitives graphiques de Caml Light fonctionnent avec n'importe quelle carte VGA en mode 320 × 200 points, 256 couleurs, mais Caml Light sait aussi profiter des possibilités supplémentaires de nombreuses cartes SuperVGA et travailler avec une meilleure résolution. Pour ce faire, vous devez déterminer quel contrôleur graphique est utilisé dans votre carte SuperVGA. Relisez la documentation fournie avec la carte, puis examinez les fichiers ayant l'extension `.GRD` ou `.GRN` (les pilotes graphiques) dans le répertoire `C:\CAML386\DEV`, et trouvez celui dont le nom semble correspondre à votre contrôleur graphique. Si vous ne savez pas quel pilote graphique utiliser, ne vous inquiétez pas : vous serez condamné au graphique VGA standard, c'est tout.

Ensuite, éditez le fichier `C:\autoexec.bat` pour :

- ajouter `C:\CAML386\BIN` à la variable `PATH`, c'est-à-dire transformer la ligne

  ```
  SET PATH=C:\DOS;...
  ```

  en

  ```
  SET PATH=C:\DOS;...;C:\CAML386\BIN
  ```

- insérer les lignes suivantes

  ```
  SET CAMLLIB=C:\CAML386\LIB
  SET GO32TMP=C:\TMP
  ```

Si votre machine possède une arithmétique flottante câblée, insérez la ligne suivante :

```
SET GO32=driver C:\CAML386\DEV\graph.grd gw 640 gh 480
```

où `graph.grd` est à remplacer par le nom du pilote graphique pour votre carte SuperVGA, déterminé ci-dessus. Les nombres `640` et `480` spécifient la résolution graphique utilisée par défaut. Vous pouvez mettre à la place `800` et `600`, ou `1024` et `768`, selon vos goûts et les capacités de votre carte.

Si vous n'avez pas pu déterminer le bon pilote graphique, n'insérez rien du tout ; laissez simplement la variable `GO32` indéfinie.

Si votre machine n'a pas d'arithmétique flottante câblée, insérez la ligne :

```
SET GO32=emu C:\CAML386\DEV\EMU387
         driver C:\CAML386\DEV\graph.grd gw 640 gh 480
```

(sur une seule ligne ; nous avons dû couper pour des raisons typographiques) où `graph.grd` est à remplacer par le nom du pilote graphique pour votre carte Super-VGA, déterminé ci-dessus. Comme nous l'avons déjà vu plus haut, vous pouvez choisir une autre résolution graphique par défaut en modifiant les nombres `640` et `480`.

Si vous n'avez pas pu déterminer le bon pilote graphique, mettez seulement :

```
SET GO32=emu C:\CAML386\DEV\EMU387
```

Puis sauvegardez le fichier `AUTOEXEC.BAT` et redémarrez la machine. Pour tester l'installation, exécutez :

```
camlc -v
```

La commande `camlc` devrait imprimer quelque chose de la forme :

```
The Caml Light system for the 80386 PC, version 0.6
  (standard library from C:\CAML386\LIB)
The Caml Light runtime system, version 0.6
The Caml Light compiler, version 0.6
The Caml Light linker, version 0.6
```

Exécutez ensuite

```
caml
```

La commande `caml` devrait afficher :

```
>       Caml Light version 0.6

#
```

En réponse au signe d'invite `#`, tapez :

```
quit();;
```

Cela devrait vous renvoyer à l'interpréteur de commandes DOS.

Problèmes. Voici quelques problèmes fréquemment rencontrés :

`Cannot find the bytecode file` (Fichier de code introuvable.)
`camlrun.exe : No such file` (Le fichier `camlrun.exe` n'existe pas.)

L'installation n'a pas été faite correctement. Vérifiez le fichier `AUTOEXEC.BAT` et la définition des variables `PATH` et `CAMLLIB`. N'oubliez pas qu'il faut redémarrer la machine pour que les modifications apportées à `AUTOEXEC.BAT` prennent effet.

`CPU must be a 386 to run this program`
(L'unité centrale doit être un 386 pour exécuter ce programme.)

Clairement, vous devrez vous contenter de la version PC 8086, car votre machine n'a pas un processeur 80386 ou 80486.

`CPU must be in REAL mode (not V86 mode)`
(L'unité centrale doit être en mode réel, et non pas en mode V86.)

Voilà un problème plus délicat. Un certain nombre de programmes utilitaires mettent le processeur 80386 dans un mode particulier, appelé mode « 86 virtuel »

(« virtual 86 » ou « V86 »), qui empêche le fonctionnement des programmes tournant en mode 32 bits protégé. Il se trouve justement que la version PC 80386 de Caml Light tourne en mode 32 bits protégé. Les utilitaires qui provoquent ce syndrome sont en particulier :

- certains pilotes de périphériques qui fournissent des services de gestion mémoire ;
- certains pilotes de périphériques utilisés en conjonction avec des programmes de mise au point (en anglais *debuggers*), comme par exemple `tdh386.sys` pour Turbo Debugger.

La version PC 80386 ne peut pas démarrer quand l'un quelconque de ces programmes est actif. Pour résoudre le problème, ne lancez pas Windows et supprimez les pilotes de périphériques coupables de votre fichier `CONFIG.SYS`.

En revanche, la version PC 80386 sait comment cohabiter avec les environnements gestionnaires de mémoire aux standard VCPI ou DPMI. C'est vrai en particulier de Windows3, de Desqview et des gestionnaires de mémoire QEMM386 et 386MAX. De même `EMM386.EXE`, distribué avec MS-DOS, fonctionne parfaitement, pourvu que vous ne l'appeliez pas avec l'option `NOEMS`. Si vous faites tourner votre PC 80386 sous un gestionnaire mémoire au standard VCPI, configurez le gestionnaire mémoire pour qu'il alloue au moins 1M de EMS, de préférence 2M (ou plus encore).

Caml Light tourne lentement et fait beaucoup d'accès disque

Quand Caml Light ne peut pas allouer la mémoire dont il a besoin, il commence à paginer sur un fichier du disque dur, ce qui ralentit considérablement l'exécution. Pour éviter ce phénomène, assurez-vous que Caml Light dispose d'au moins 1M de mémoire (de préférence 2M). Caml Light utilise la mémoire EMS si vous employez un gestionnaire mémoire au standard VCPI, et la mémoire XMS sinon. Si vous employez un gestionnaire mémoire VCPI, vérifiez sa configuration pour être sûr que Caml Light peut allouer assez de mémoire EMS.

13
Bibliographie

Pour le lecteur qui désire approfondir certains aspects du langage ML, nous indiquons ici quelques livres et rapports de recherche sur ce sujet.

13.1 Programmer en ML

Les livres suivants sont des cours de programmation dont le support est ML. Sans pour autant décrire ML de manière exhaustive, ils fournissent de bonnes introductions au langage ML. Certains sont écrits en Standard ML et non pas en Caml.

- Pierre Weis et Xavier Leroy, *Le langage Caml*, InterÉditions, 1993.

  Le complément naturel de ce manuel. Une initiation progressive à la programmation en Caml. On y présente de nombreux exemples de programmes Caml réalistes.

- Lawrence C. Paulson, *ML for the working programmer*, Cambridge University Press, 1991.

  Une bonne introduction à la programmation en Standard ML avec plusieurs exemples complets, dont un démonstrateur de théorèmes. Présente également le système de modules de Standard ML.

- Chris Reade, *Elements of functional programming*, Addison-Wesley, 1989.

  Une introduction à Standard ML, avec des parties sur la sémantique dénotationnelle et le lambda-calcul.

- Ryan Stansifer, *ML primer*, Prentice Hall, 1992.

  Une introduction brève, mais bien faite, à la programmation en Standard ML.

- Thérèse Accart Hardin et Véronique Donzeau-Gouge Viguié, *Concepts et outils de la programmation. Du fonctionnel à l'impératif avec Caml et Ada*, InterÉditions, 1992.

  Un premier cours de programmation, qui introduit certaines notions de programmation, d'abord en Caml, puis en Ada. S'adresse aux débutants, un peu lent pour les autres.

- Ake Wikstrom, *Functional programming using Standard ML*, Prentice Hall, 1987.

Un premier cours de programmation, enseigné en Standard ML. À l'intention des tout débutants, lent pour les autres.

- Harold Abelson et Gerald Jay Sussman, *Structure and interpretation of computer programs*, The MIT Press, 1985. (Traduction française : *Structure et interprétation des programmes informatiques*, InterÉditions, 1989.)

  Un cours de programmation remarquable, enseigné en Scheme, le dialecte moderne de Lisp. Mérite d'être lu même si vous vous intéressez plus à ML qu'à Lisp.

13.2 Descriptions des dialectes de ML

Les livres et rapports de recherche qui suivent sont des descriptions de différents langages de programmation de la famille ML. Tous supposent une certaine familiarité avec ML.

- Robert Harper, *Introduction to Standard ML*, Technical report ECS-LFCS-86-14, Université d'Édimbourg, 1986.

  Un aperçu de Standard ML, système de modules inclus. Dense, mais encore lisible.

- Robin Milner, Mads Tofte et Robert Harper, *The definition of Standard ML*, The MIT Press, 1990.

  Une définition formelle complète de Standard ML, dans le cadre de la sémantique opérationnelle structurée. Ce livre est sans doute la définition d'un langage de programmation la plus mathématiquement précise jamais écrite. Pour lecteurs connaissant parfaitement ML et que les mathématiques n'effraient pas.

- Robin Milner et Mads Tofte, *Commentary on Standard ML*, The MIT Press, 1991.

  Un commentaire sur le livre précédent, qui tente d'expliquer les parties les plus délicates et de motiver les choix conceptuels. Un peu moins difficile à lire que la Définition.

- Guy Cousineau et Gérard Huet, *The CAML primer*, Rapport technique 122, INRIA, 1990. (Disponible par FTP anonyme sur `ftp.inria.fr`, répertoire `lang/caml/V2-6.1`, fichier `doc.tar.Z`.)

  Une brève description du premier système Caml, dont Caml Light est issu. Suppose une certaine familiarité avec Lisp.

- Pierre Weis *et al.*, *The CAML reference manual, version 2.6.1*, Rapport technique 121, INRIA, 1990. (Disponible par FTP anonyme sur `ftp.inria.fr`, répertoire `lang/caml/V2-6.1`, fichier `doc.tar.Z`.)

  Le manuel du premier système Caml, dont Caml Light est issu.

- Michael J. Gordon, Arthur J. Milner et Christopher P. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science volume 78, Springer-Verlag, 1979.

  Ce livre est la première description publiée du langage ML, à l'époque où il n'était rien de plus que le langage de contrôle d'un démonstrateur de théorèmes, le système LCF. Ce livre est maintenant dépassé, puisque le langage ML a beaucoup évolué depuis, mais reste d'intérêt historique.

- Paul Hudak, Simon Peyton-Jones et Philip Wadler, *Report on the programming language Haskell, version 1.1*, Rapport technique, université de Yale, 1991. Également publié dans le journal *SIGPLAN Notices* de mai 1992 (volume 27, numéro 5).

  Haskell est un langage purement fonctionnel dont la sémantique est paresseuse et qui a de nombreux points communs avec ML (pleine fonctionnalité, typage polymorphe), mais possède aussi des traits propres intéressants (surcharge dynamique, aussi appelée « classes »).

13.3 Implémentation des langages de programmation fonctionnels

Les références qui suivent intéresseront ceux qui veulent savoir comment le langage ML se compile et s'implémente.

- Xavier Leroy, *The ZINC experiment : an economical implementation of the ML language*, Rapport technique 117, INRIA, 1990. (Disponible par FTP anonyme sur `ftp.inria.fr`.)

  Une description de ZINC, le prototype d'implémentation de ML qui a évolué en Caml Light. De grandes parties de ce rapport s'appliquent encore au système Caml Light, en particulier la description du modèle d'exécution et de la machine abstraite. D'autres parties sont dépassées. Malgré tout, ce rapport donne encore une bonne idée des techniques d'implémentation utilisées pour Caml Light.

- Simon Peyton-Jones, *The implementation of functional programming languages*, Prentice Hall, 1987. (Traduction française : *Mise en œuvre des langages fonctionnels de programmation*, Masson, 1990.)

  Une excellente description de l'implémentation des langages purement fonctionnels à sémantique paresseuse, à l'aide de la technique de réduction de graphe. La partie du livre qui traite des transformations de ML en lambda-calcul enrichi s'applique directement à Caml Light. Vous y trouverez une bonne description de la compilation du filtrage et de l'inférence de types.

- Andrew W. Appel, *Compiling with continuations*, Cambridge University Press, 1992.

  Une description complète d'un compilateur optimisant pour Standard ML, reposant sur une représentation intermédiaire appelée « style à passage de continuations » (en anglais *continuation-passing style*). On y montre comment appliquer à ML de nombreuses optimisations de programme.

13.4 Applications de ML

Les rapports qui suivent illustrent l'utilisation de ML dans des domaines sortant nettement de ses premières applications (démonstration automatique et compilation).

- Emmanuel Chailloux et Guy Cousineau, *The MLgraph primer*, Rapport technique 92-15, École Normale Supérieure, 1992. (Disponible par FTP anonyme sur `ftp.ens.fr`.)

  Ce rapport décrit une bibliothèque Caml Light pour produire des images Postscript par manipulations de haut niveau.

- Xavier Leroy, *Programmation du système Unix en Caml Light*, Rapport technique 147, INRIA, 1992. (Disponible par FTP anonyme sur `ftp.inria.fr`.)

  Un cours de programmation du système Unix, montrant l'utilisation de la bibliothèque Caml Light donnant accès aux appels système Unix.

- John H. Reppy, *Concurrent programming with events — The concurrent ML manual*, Cornell University, 1990. (Disponible par FTP anonyme sur `research.att.com`.)

  Concurrent ML ajoute à Standard ML of New Jersey la possibilité de programmer plusieurs processus s'exécutant en parallèle et communiquant *via* des canaux de communication.

- Jeannette M. Wing, Manuel Faehndrich, J. Gregory Morrisett et Scottt Nettles, *Extensions to Standard ML to support transactions*, Rapport technique CMU-CS-92-132, Carnegie-Mellon University, 1992. (Disponible par FTP anonyme sur `reports.adm.cs.cmu.edu`.)

  Comment intégrer à Standard ML les opérations principales des bases de données.

- Emden R. Gansner et John H. Reppy, *eXene*, Bell Labs, 1991. (Disponible par FTP anonyme sur `research.att.com`.)

  Une interface entre Standard ML of New Jersey et le système de fenêtrage X Windows.

Index